COMPILER DESIGN PROJECT REPORT - 4

INTERMEDIATE CODE GENERATION FOR C LANGUAGE

Submitted By :

Arvind Ramachandran - 15CO111

Aswanth P P - 15CO112

DATE : 28/03/2018

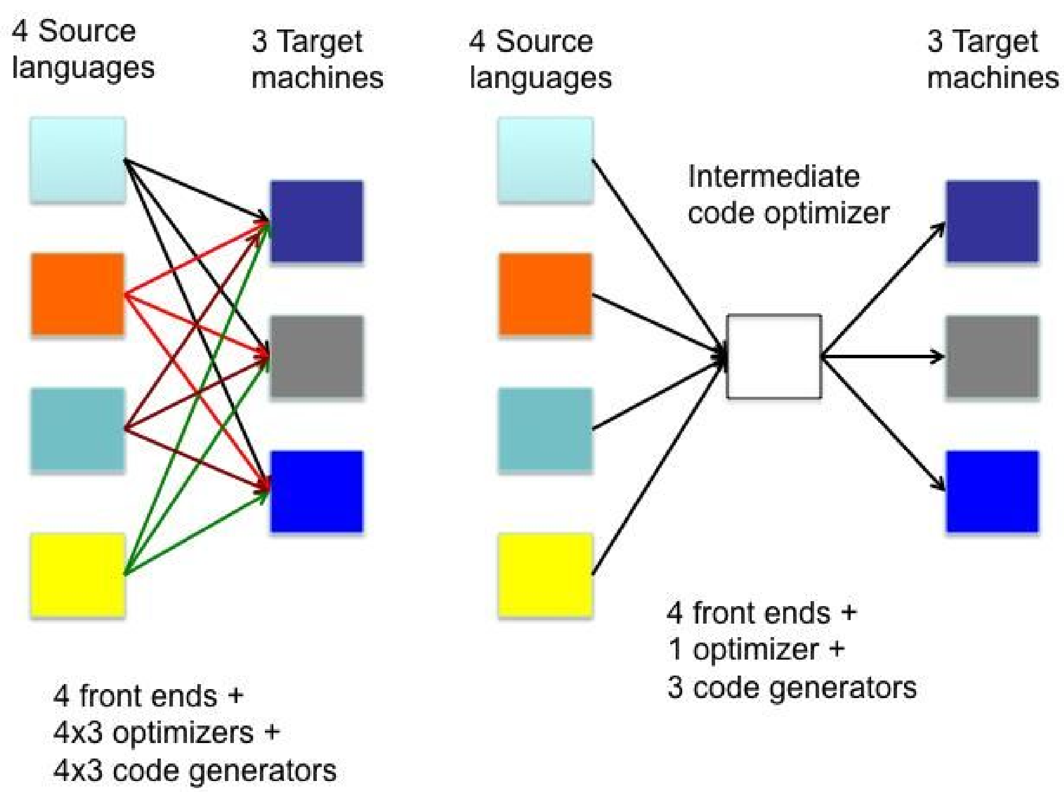
TABLE OF CONTENTS

INTRODUCTION

Compilers generate machine code, whereas interpreters interpret intermediate code. Interpreters are easier to write and can provide better error messages (symbol table is still available). However, they are at least 5 times slower than machine code generated by compilers and also require much more memory than machine code generated by compilers.

While generating machine code directly from source code is possible, it entails two problems :

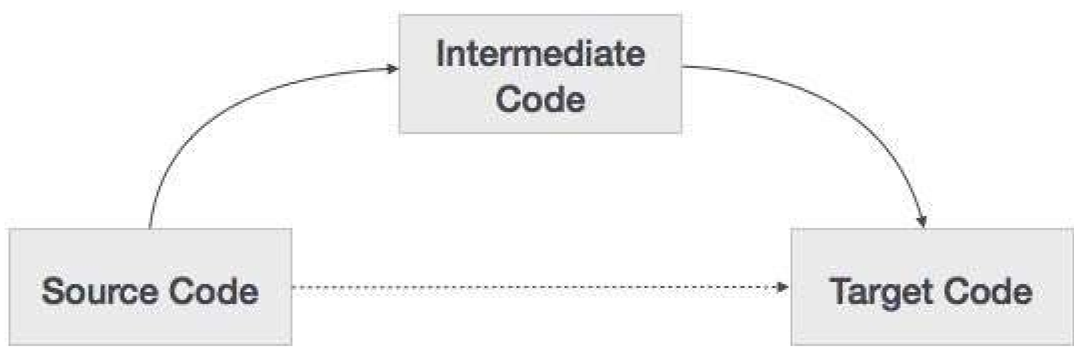
1) With **m** languages and n target machines, we need to write **m** front ends, **m** × **n** optimizers, and **m** × **n** code generators
2) The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused.



By converting source code to an intermediate code, a machine-independent code optimizer may be written. This means just **m** front ends, **n** code generators and **1** optimizer.

A source code can directly be translated into its target machine code, but we need to translate the source code into an intermediate code which is then translated to its target code. The main reasons for this are :

1) If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
2) Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
3) The second part of compiler, synthesis, is changed according to the target machine.
4) It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.



Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

INTERMEDIATE REPRESENTATION

The aim of Intermediate Code Generation is to translate the program into a format expected by the compiler back-end. Intermediate code must be easy to produce and easy to translate to machine code.

Why use an intermediate representation ?

1) It's easy to change the source or the target language by adapting only the front-end or back-end (portability).

2) It makes optimization easier: one needs to write optimization methods only for the intermediate representation.

3) The intermediate representation can be directly interpreted.

Intermediate codes can be represented in a variety of ways and they have their own benefits.

1) **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
2) **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Some general forms of intermediate representation are :

1) Graphical IR (parse tree, abstract syntax trees, DAG. . .)

2) Linear IR (ie., non graphical)

3) Three Address Code (TAC): instructions of the form "result=op1 operator op2"

4) Static single assignment (SSA) form: each variable is assigned once

5) Continuation-passing style (CPS): general form of IR for functional language

THREE-ADDRESS CODE

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b * c + d :

- T 1 = b * c
- T 2 = a + T 1
- T 3 = T 2 + d

where T 1 , T 2 , T 3 are temporary variables.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

1) **Quadruples** - Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. Consider the example, ***a = b + c * d;*** It is represented below in quadruples format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

2) **Triples** - Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg1	arg2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

3) **Indirect Triples -** This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

IMPLEMENTATION

When generating IR at this level, do not need to worry about optimizing it.It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc.

All the lexical analyzer part are done in `lexicalAnalyzer.l` which returns each token along with its value or type based on whether it is variable or constant or keyword. Syntax and semantic check has been done in `syntaxChecker.y.` All the productions and respective semantic actions are written here .

The functions that are generating intermediate code has been written in `icg{.h,.c}.` The main functions in icg files are

- `threeAddressCode* appendCode(char *code)`

  Append the three address code provided as argument to the main set of three addresscode

- `void backpatch(backPatchList * list, int gotoL)`

  It is to add the line number “gotoL” to backPatchList

- `backPatchList* mergelists(backPatchList  * a, backPatchList * b)`

  It is to merge to back patch list if it is resolved.

- `backPatchList* appendToBackPatch(backPatchList  * list, threeAddressCode  * entry)`

  It is to append a single line three address code entry to the back patch list.

- `tokenList* appendToSymbolTable(char *name,tokenType type,tokenReturnType returnType,long size,long line,char *scope,long parameter)`

  It is used add token "name" to the symbol table along with all the properties

- `void writeCode(FILE *icgOut)`

  This is write the three address code generated till now into a file provided as argument

- `void writeSymbolTable(FILE *symOut)`

  This is to write the symbol table entries till now to the file provided as argument

These functions are called from respective productions in `syntaxAnalyzer.y` whenever needed.All the code returned are stored in a global pointer such that can write it to a file at the end. The source code for the same has been provided below.

SOURCE CODE

lexicalAnalyzer.l

This is the lex program that contains regular expressions and returns whatever tokens are required.

```
%option noyywrap
%option nomain
%option yylineno
%{
#include "icg.h"
#include "y.tab.h"
int f=0;
%}
NEWLINE (\n|\n\l|\r\n)
WHITESPACE (" "|\t)
SINGLELINE_COMMENT ("//".*{NEWLINE})
MULTILINE_COMMENT ("/*".*"*/")
PREPROCESSOR ("#".*{NEWLINE})
INTEGER (0|[1-9][0-9]*)
FLOAT (0\.0|[1-9][0-9]*\.[0-9]+)
IDENTIFIER ([a-zA-Z_][a-zA-Z0-9_]*)
LIBFUNC    "printf"|"scanf"
%%
({NEWLINE}+|{WHITESPACE}+) {}
({SINGLELINE_COMMENT}|{MULTILINE_COMMENT}) {}
{PREPROCESSOR} {}
^("+"|"-") {
                if('+' == yytext[0]) {
                        return(U_PLUS);
                } else {
                        return(U_MINUS);
                }
                f = 0;
        }
("("|")"|"{"|"}"|"<"|">"|"="|";"|","|"!")       {f = 1;return(yytext[0]);}
("/"|"*"|"%")                                   {f = 1;return(yytext[0]);}
("+"|"-") {
                if(f==1) {
                        if('+' == yytext[0]) {
                        return(U_PLUS);
                }
                else {
                        return(U_MINUS);
                }
                f = 0;
                } else {
                        return(yytext[0]);
                }
        }

"void"                                          {return(VOID);}
"int"                                           {return(INT);}
"float"                                         {return(FLOAT);}
"constant"                                      {return(CONSTANT);}
"if"                                            {return(IF);}
"else"                                          {return(ELSE);}
"return"                                        {return(RETURN);}
"do"                                            {return(DO);}
"while"                                         {return(WHILE);}
"for"                                           {return(FOR);}
"++"                                            {return(INC_OP);}
"--"                                            {return(DEC_OP);}
"=="                                            {return(EQUAL);}
"!="                                            {return(NOT_EQUAL);}
">="                                            {return(GREATER_OR_EQUAL);}
"<="                                            {return(LESS_OR_EQUAL);}
"<<"                                            {return(SHIFTLEFT);}
"&&"                                            {return(LOG_AND);}
"||"                                            {return(LOG_OR);}
{INTEGER} {
                yylval.expr.value = strdup(yytext);
                yylval.expr.cType = CONST_type;
                yylval.expr.type = INT_type;
                f = 0;
                return(CONSTANT);
        }
{FLOAT} {
                yylval.expr.value = strdup(yytext);
                yylval.expr.cType = CONST_type;
                yylval.expr.type = FLOAT_type;
                f = 0;
                return(CONSTANT);
        }
{IDENTIFIER} {
                yylval.str = malloc(strlen(yytext)*sizeof(char)+1);
                strcpy(yylval.str, yytext);
                f = 0;
                return(IDENTIFIER);
        }

%%
```

syntaxChecker.y

This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

```
%{
#include "icg.h"
extern int yylineno;
extern char *yytext;
struct label{
        char *funcName;
        int label;
        struct label *next;
};
struct label *labelHead=NULL;

int label[10];
int funcCount=0;
int paramCount;
char icgQuad[50];
int funcLineNumber = 0;
%}
%token VOID INT FLOAT CONSTANT IDENTIFIER
%token IF ELSE RETURN DO WHILE FOR
%token INC_OP DEC_OP U_PLUS U_MINUS
%token EQUAL NOT_EQUAL GREATER_OR_EQUAL LESS_OR_EQUAL SHIFTLEFT LOG_AND LOG_OR

%right '='
%left LOG_OR
%left LOG_AND
%left '<' '>' LESS_OR_EQUAL GREATER_OR_EQUAL
%left EQUAL NOT_EQUAL
%left SHIFTLEFT
%left '+' '-'
%left '*' '/' '%'
%right U_PLUS U_MINUS '!'
%left INC_OP DEC_OP
%union
{
    char                *str;
    int                 integer;
    float               real;
    int                 type;
        struct
        {
            char                        *value;
            int                         type;
            int                         cType;
            struct BackpatchList*       trueList;
            struct BackpatchList*       falseList;
```

```
        } expr;
        struct
        {
          struct BackpatchList*         nextList;
        } stmt;
        struct
        {
          int                           quad;
          struct BackpatchList*         nextList;
        } mark;
        struct
        {
            int                         count;
            struct tokenList *  queue;
        } exp_list;
}

%type <str> id IDENTIFIER
%type <type> declaration var_type
%type <expr> expression assignment CONSTANT
%type <stmt> statement statement_list matched_statement unmatched_statement program function_body function
%type <exp_list> exp_list
%type <mark> marker jump_marker
%start program_head
%%
program_head
    : program
        {
            tokenList * mainFunc = getSymbol("main");
            if(mainFunc == NULL){
                printf("ERROR: Main function not found!\n");
                yyerror();
            }
            backpatch($1.nextList,mainFunc->line+1);
        }
    ;
program
    : jump_marker function
        {
            $$.nextList = $1.nextList;
            backpatch($2.nextList, nextquad());
        }
    | program function
        {
            $$.nextList = $1.nextList;
```

```
            backpatch($2.nextList, nextquad());
        }
    ;
function
    : var_type id '(' parameter_list ')' ';'
        {
            if(labelHead==NULL){
                struct label *temp=(struct label *)malloc(sizeof(label));
                temp->name=strdup(id);
                temp->label=funcCount;
                temp->next=NULL;
            }
            else{
                struct label *temp=(struct label *)malloc(sizeof(label));
                temp->name=strdup(id);
                temp->label=funcCount;
                temp->next=NULL;
                struct label *p=labelHead;
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next=temp;
            }

            funcCount++;
            addFunctionPrototype($2, paramCount, $1);
            paramCount = 0;
            $$.nextList = NULL;
        }
    | var_type id '(' parameter_list ')' function_body
        {

            addFunction($2, paramCount, $1, funcLineNumber);
            paramCount = 0;
            funcLineNumber = nextquad();
            $$.nextList = $6.nextList;
        }
    ;
function_body
    : '{' statement_list  '}'
        {
            $$.nextList = $2.nextList;
        }
    | '{' declaration_list statement_list '}'
        {
```

```
            $$.nextList = $3.nextList;
        }
    ;
declaration_list
    : declaration ';'
    | declaration_list declaration ';'
    ;
declaration
    : INT id
        {
            $$ = INT_type;
            addSymbolToQueue($2, INT_type, 0);
        }
    | FLOAT id
        {
            $$ = FLOAT_type;
            addSymbolToQueue($2, FLOAT_type, 0);
        }
    | declaration ',' id
        {
            if(INT_type == $1) {
                addSymbolToQueue($3, INT_type, 0);
            } else if(FLOAT_type == $1) {
                addSymbolToQueue($3, FLOAT_type, 0);
            }
        }
    ;
parameter_list
    : INT id
        {
            paramCount++;
            addSymbolToQueue($2, INT_type, paramCount);
        }
    | FLOAT id
        {
            paramCount++;
            addSymbolToQueue($2, FLOAT_type, paramCount);
        }
    | parameter_list ',' INT id
        {
            paramCount++;
            addSymbolToQueue($4, INT_type, paramCount);
        }
    | parameter_list ',' FLOAT id
        {
```

```
            paramCount++;
            addSymbolToQueue($4, FLOAT_type, paramCount);
        }
    | VOID
    |
    ;
var_type
    : VOID
        {
            $$ = Return_VOID;
        }
    | INT
        {
            $$ = Return_INT;
        }
    | FLOAT
        {
            $$ = Return_FLOAT;
        }
    ;
statement_list
    : statement
        {
            $$.nextList = $1.nextList;
        }
    | statement_list marker statement
        {
            backpatch($1.nextList,$2.quad);
            $$.nextList = $3.nextList;
        }
    ;
statement
    : matched_statement
        {
            $$.nextList = $1.nextList;
        }
    | unmatched_statement
        {
            $$.nextList = $1.nextList;
        }
    ;
matched_statement
    : IF '(' assignment ')' marker matched_statement jump_marker ELSE marker matched_statement
        {
            backpatch($3.trueList,$5.quad);
```

```
            backpatch($3.falseList,$9.quad);
            $$.nextList = mergelists($7.nextList,$10.nextList);
            $$.nextList = mergelists($$.nextList,$6.nextList);
        }
    | assignment ';'
        {
            $$.nextList = NULL;
        }
    | RETURN ';'
        {
            $$.nextList = NULL;
            sprintf(icgQuad,"RETURN");
            appendCode(icgQuad);
        }
    | RETURN assignment ';'
        {
            $$.nextList = NULL;
            sprintf(icgQuad,"RETURN %s",$2.value);
            appendCode(icgQuad);
        }
    | WHILE marker '(' assignment ')' marker matched_statement jump_marker
        {
            backpatch($4.trueList,$6.quad);
            $$.nextList = $4.falseList;
            backpatch($7.nextList,$2.quad);
            backpatch($8.nextList,$2.quad);
        }
    | DO marker statement WHILE '(' marker assignment ')' ';'
        {
            backpatch($3.nextList,$6.quad);
            backpatch($7.trueList,$2.quad);
            $$.nextList = $7.falseList;
        }
    | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker matched_statement jump_marker
        {
            if(BOOL_type == $3.type || BOOL_type == $9.type) {
                printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
                yyerror();
            }
            if(BOOL_type != $6.type) {
                printf("error, 2nd argument of for loop must be boolean\n");
                yyerror();
            }
            backpatch($3.trueList, $5.quad);
            backpatch($13.nextList, $8.quad);
```

```
            backpatch($14.nextList, $8.quad);
            $$.nextList = $6.falseList;
            backpatch($6.trueList, $12.quad);
            backpatch($9.trueList, $5.quad);
            backpatch($10.nextList, $5.quad);

        }
    | '{' statement_list '}'
        {
            $$.nextList = $2.nextList;
        }
    | '{' '}'
        {
            $$.nextList = NULL;
        }
    ;
unmatched_statement
    : IF '(' assignment ')' marker statement
        {
            backpatch($3.trueList,$5.quad);
            $$.nextList = mergelists($3.falseList,$6.nextList);
        }
    | WHILE marker '(' assignment ')' marker unmatched_statement jump_marker
        {
            backpatch($4.trueList,$6.quad);
            $$.nextList = $4.falseList;
            backpatch($7.nextList,$2.quad);
            backpatch($8.nextList,$2.quad);
        }
    | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker unmatched_statement jump_marker
        {
            if(BOOL_type == $3.type || BOOL_type == $9.type) {
                printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
                yyerror();
            }
            if(BOOL_type != $6.type) {
                printf("error, 2nd argument of for loop must be boolean\n");
                yyerror();
            }
            backpatch($3.trueList, $5.quad);
            backpatch($13.nextList, $8.quad);
            backpatch($14.nextList, $8.quad);
            $$.nextList = $6.falseList;
            backpatch($6.trueList, $12.quad);
            backpatch($9.trueList, $5.quad);
            backpatch($10.nextList, $5.quad);
        }
    | IF '(' assignment ')' marker matched_statement jump_marker ELSE marker unmatched_statement
        {
            backpatch($3.trueList,$5.quad);
            backpatch($3.falseList,$9.quad);
            $$.nextList = mergelists($7.nextList,$10.nextList);
            $$.nextList = mergelists($$.nextList,$6.nextList);
        }
    ;
assignment
    : expression
        {
            $$=$1;
        }
    | id '=' expression
        {
            int destType = getSymbolType($1);
                if(destType == 0){
                        printf("ERROR: Not in scope");
                }
            if(destType != $3.type) {
                printf("Type error on line: %d\n", yylineno);
                yyerror();
            }

            sprintf(icgQuad,"%s := %s",$1,$3.value);
            appendCode(icgQuad);
            $$.type = destType;
            $$.trueList = $3.trueList;
            $$.cType = VAR_type;
            $$.value = $1;
        }
    ;
expression
    : INC_OP expression
        {
            if($2.type != INT_type){
                    printf("ERROR: Increment not allowed for types different than Integer.\n");
                    yyerror();
            }
            //Create a variable if needed
            if($2.cType != VAR_type){
                    char *var = nextIntVar();
                    sprintf(icgQuad,"%s := %s",var,$2.value);
```

```
                appendCode(icgQuad);
                free($2.value);
                $2.value = var;
                $2.type = INT_type;
                $2.cType = VAR_type;
        }
        sprintf(icgQuad,"%s := %s + 1",$2.value,$2.value);
        appendCode(icgQuad);
        //Set the attributes
        $$ = $2;
        $$.trueList = NULL;
        $$.falseList = NULL;
    }
  | DEC_OP expression
    {
        if($2.type != INT_type){
                printf("ERROR: Decrement not allowed for types different than Integer.\n");
                yyerror();
        }
        //Create a variable if needed
        if($2.cType != VAR_type){
                char *var = nextIntVar();
                sprintf(icgQuad,"%s := %s",var,$2.value);
            appendCode(icgQuad);
                free($2.value);
                $2.value = var;
                $2.type = INT_type;
                $2.cType = VAR_type;
        }
        sprintf(icgQuad,"%s := %s - 1",$2.value,$2.value);
        appendCode(icgQuad);
        //Set the attributes
        $$ = $2;
        $$.trueList = NULL;
        $$.falseList = NULL;
    }
  | expression LOG_OR marker expression
    {
        if(BOOL_type != $1.type) {
            sprintf(icgQuad, "IF (%s <> 0) GOTO", $1.value);
            $1.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad, "GOTO");
            $1.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        }
        if(BOOL_type != $4.type) {
            sprintf(icgQuad, "IF (%s <> 0) GOTO", $4.value);
            $4.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad, "GOTO");
            $4.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        }
        $$.trueList = mergelists($1.trueList, $4.trueList);
        backpatch($1.falseList, $3.quad);
        $$.falseList = $4.falseList;
        $$.type = BOOL_type;
    }
  | expression LOG_AND marker expression
    {
        if(BOOL_type != $1.type) {
            sprintf(icgQuad, "IF (%s <> 0) GOTO", $1.value);
            $1.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad, "GOTO");
            $1.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        }
        if(BOOL_type != $4.type) {
            sprintf(icgQuad, "IF (%s <> 0) GOTO", $4.value);
            $4.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad, "GOTO");
            $4.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        }
        $$.falseList = mergelists($1.falseList, $4.falseList);
        backpatch($1.trueList, $3.quad);
        $$.trueList = $4.trueList;
        $$.type = BOOL_type;
     }
  | expression NOT_EQUAL expression
    {

        if($1.type != INT_type && $1.type != FLOAT_type){
            printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
            yyerror();
        }
        sprintf(icgQuad,"IF (%s <> %s) GOTO",$1.value,$3.value);
        $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
        sprintf(icgQuad,"GOTO");
        $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        $$.value = "TrueFalse Only!";
        $$.type = BOOL_type;
        $$.cType = NONE_type;

    }
```

```
    | expression EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s = %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        if(BOOL_type == $1.type) {
            $$.trueList = mergelists($$.trueList, $1.trueList);
            $$.falseList = mergelists($$.falseList, $1.falseList);
        }
        if(BOOL_type == $3.type) {
            $$.trueList = mergelists($$.trueList, $3.trueList);
            $$.falseList = mergelists($$.falseList, $3.falseList);
        }
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression GREATER_OR_EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s >= %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression LESS_OR_EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s <= %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
```

```
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '>' expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s > %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '<' expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s < %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '+' expression
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when adding numbers.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
```

```
            }

            char* var = NULL;
            switch(type){
                case INT_type: var = nextIntVar();break;
                case FLOAT_type:var = nextFloatVar();break;
            }
            char buffer[50];
            sprintf(icgQuad,"%s := %s + %s",var,$1.value,$3.value);
            appendCode(icgQuad);
            $$.value = var;
            $$.type = type;
            $$.cType = VAR_type;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | expression '-' expression
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when substracting numbers.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
            }

            char* var = NULL;
            switch(type){
                case INT_type: var = nextIntVar();break;
                case FLOAT_type:var = nextFloatVar();break;
            }
            char buffer[50];
            sprintf(icgQuad,"%s := %s - %s",var,$1.value,$3.value);
            appendCode(icgQuad);
            $$.value = var;
            $$.type = type;
            $$.cType = VAR_type;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | expression '*' expression
```

```
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when multiplicating numbers.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
            }
            char* var = NULL;
            switch(type){
                case INT_type: var = nextIntVar();break;
                case FLOAT_type:var = nextFloatVar();break;
            }
            char buffer[50];
            sprintf(icgQuad,"%s := %s * %s",var,$1.value,$3.value);
            appendCode(icgQuad);
            $$.value = var;
            $$.type = type;
            $$.cType = VAR_type;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }

    | expression '/' expression
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when dividing numbers.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
            }
            char* var = NULL;
            switch(type){
                case INT_type: var = nextIntVar();break;
                case FLOAT_type:var = nextFloatVar();break;
            }
            char buffer[50];
```

```
            sprintf(icgQuad,"%s := %s / %s",var,$1.value,$3.value);
            appendCode(icgQuad);
            $$.value = var;
            $$.type = type;
            $$.cType = VAR_type;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | expression '%' expression
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when caluclating mod.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
            }
            char* var = NULL;
            switch(type){
                case INT_type: var = nextIntVar();break;
                case FLOAT_type:var = nextFloatVar();break;
            }
            char buffer[50];
            sprintf(icgQuad,"%s := %s \% %s",var,$1.value,$3.value);
            appendCode(icgQuad);
            $$.value = var;
            $$.type = type;
            $$.cType = VAR_type;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | '!' expression
        {
            if($2.type != BOOL_type){
                if($2.type != INT_type && $2.type != FLOAT_type){
                    printf("ERROR: Only Bool, Int and Float allowed in logical expressions!\n");
                    yyerror();
                }
                sprintf(icgQuad,"IF (%s <> 0) GOTO",$2.value);
                $$.falseList = appendToBackPatch(NULL,appendCode(icgQuad));
                sprintf(icgQuad,"GOTO",$2.value);
```

```
                $$.trueList = appendToBackPatch(NULL,appendCode (icgQuad));
            }
            else{
              $$ = $2;
              $$.trueList = $2.falseList;
              $$.falseList = $2.trueList;
            }
        }
    | U_PLUS expression
        {
            if(INT_type != $2.type && FLOAT_type != $2.type) {
                yyerror();
            }
            $$ = $2;
        }
    | U_MINUS expression
        {
            $$ = $2;
            if(INT_type == $2.type) {
                $$.value = nextIntVar();
            } else if (FLOAT_type == $2.type) {
                $$.value = nextFloatVar();
            } else {
                yyerror();
            }
            sprintf(icgQuad, "%s := -%s", $$.value, $2.value);
            appendCode (icgQuad);
       }
    | CONSTANT
        {
            $$.value = strdup(yytext);
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | '(' expression ')'
        {
            $$ = $2;
        }
    | id '(' exp_list ')'
        {
            int varType = getFunctionType($1);
            if(varType == 0){
                printf("ERROR: Function %s not defined!\n",$1);
                yyerror();
            }
```

```
            char* var = NULL;
            switch(varType){
            case Return_INT:
                var = nextIntVar();
                $$.type = INT_type;
                break;
            case Return_FLOAT:
                var = nextFloatVar();
                $$.type = FLOAT_type;
                break;
            }
            $$.value = var;
            $$.cType = VAR_type;
            checkAndGenerateParams($3.queue,$1,$3.count);
            sprintf(icgQuad,"%s := CALL %s, %d",var,$1,$3.count);
            appendCode (icgQuad);

        }
    | id '('  ')'
        {
            int varType = getFunctionType($1);
            if(varType == 0){
                printf("ERROR: Function %s not defined!\n",$1);
                yyerror();
            }
            char* var = NULL;
            switch(varType){
                case Return_INT:
                        var = nextIntVar;
                        $$.type = INT_type;
                        break;
                case Return_FLOAT:
                        var = nextFloatVar;
                        $$.type = FLOAT_type;
                break;

            }
            $$.value = var;
            $$.cType = VAR_type;
            checkAndGenerateParams(NULL,$1,0);
            sprintf(icgQuad,"%s := CALL %s, %d",var,$1,0);
            appendCode (icgQuad);
        }
    | id {
            int varType = getSymbolType($1);
```

```
            if(varType == 0){
                printf("ERROR: Variable %s not in scope!\n",$1);
                yyerror();
            }
            $$.value = $1;
            $$.type = varType;
            $$.cType = VAR_type;
        }
    ;
exp_list
    : expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer and Float are allowed as parameter types.\n");
                yyerror();
            }
            $$.queue = addSymbolToParameterQueue(NULL,$1.value,$1.type);
            $$.count = 1;
        }
    | exp_list ',' expression
        {
              if($3.type != INT_type && $3.type != FLOAT_type){
                  printf("ERROR: Only Integer and Float are allowed as parameter types.\n");
                  yyerror();
              }
              $$.queue = addSymbolToParameterQueue($1.queue,$3.value,$3.type);
              $$.count = $1.count + 1;
        }
    ;
id : IDENTIFIER
        {
            $$ = strdup(yytext);
        }
    ;
marker
        : {
              $$.quad = nextquad();
        };
jump_marker
        : {
              $$.quad = nextquad();
              sprintf(icgQuad,"GOTO");
              $$.nextList = appendToBackPatch(NULL, appendCode (icgQuad));
    };
%%
```

icg.c

```c
#include "icg.h"
#include "y.tab.h"

tokenList  *symPtr = NULL;
tokenList  *bufferPtr = NULL;
threeAddressCode   *codePtr = NULL;

extern int yylineno;
extern int yyin;

int globalOffset = 0, tempIntCounter = 0,tempFloatCounter= 0,currentLine = -1;

void yyerror() {
    printf("ERROR: %d : Syntactical Error\n",yylineno);
    exit(1);
}

int main(int argc, char **argv) {
    yyin = fopen(argv[1],"r");
    yyparse();
    printf("\n\n\t Parsing Completed\n");
    printf("Three Address Code and Symbol table are generated\n\n");
    FILE *icgPtr= fopen("threeAddressCode.txt","w");
    writeCode(icgPtr);
    FILE *symPtr = fopen("symbolTable.txt", "w");
    writeSymbolTable(symPtr);
    fclose(icgPtr);
    fclose(symPtr);
    printf("\n");
    return(0);
}

static const char* returnTypeToString(tokenReturnType type)
{
    switch(type)
    {
        case Return_VOID:
            return "None";

        case Return_INT:
            return "Int";

        case Return_FLOAT:
            return "Real";
    }
}
```

```c
}

static const char* dataTypeToString(tokenType type)
{
    switch(type)
    {

        case INT_type:
            return "Int";

        case FLOAT_type:
            return "Real";

         case BOOL_type:
          return "Bool";

        case FUNC:
            return "Func";
        case PROTO:
            return "Proto";

        case MAIN:
            return "Main";
    }
}

threeAddressCode* appendCode(char *code){

        threeAddressCode *temp = malloc(sizeof(threeAddressCode ));

        temp->code=(char *)malloc(sizeof(char)*strlen(code));
        strcpy(temp->code,code);
        temp->gotoLine = -1;
        temp->next = NULL;


        if(codePtr == NULL){
                codePtr=temp;
        }
        else{
                threeAddressCode *p=codePtr;
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next=temp;
```

```c
        }
        currentLine++;
        return temp;
}

void backpatch(backPatchList* list, int gotoL){
        if(list == NULL){
                return;
        } else{
                backPatchList* temp;
                while(list){
                        if(list->entry != NULL){
                                list->entry->gotoLine = gotoL;
                        }
                        //printf("backpatching: %s",list->entry->code);
                        temp = list;
                        list = list->next;
                        free(temp);
                }
        }
}

backPatchList* mergelists(backPatchList* a, backPatchList* b){
        if(a != NULL && b == NULL){
                return a;
        }
        else if(a == NULL && b != NULL){
                return b;
        }
        else if(a == NULL && b == NULL){
                return NULL;
        }
        else{
                backPatchList* temp = a;
                while(a->next){
                        a = a->next;
                }
                a->next = b;
                return temp;
        }
}

backPatchList* appendToBackPatch(backPatchList* p, threeAddressCode* newCode){

        if(newCode == NULL){
                return p;
        }
        else if(p == NULL){
                backPatchList* temp = malloc(sizeof(backPatchList));
                temp->entry = newCode;
                temp->next = NULL;
                return temp;
        }
        else{
                backPatchList* temp = malloc(sizeof(backPatchList));
                temp->entry = newCode;
                temp->next=NULL;
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next = temp;
                return temp;
        }
}

tokenList* appendToSymbolTable(char *name,tokenType type,tokenReturnType returnType,long size,long line,char *scope,long parameter)
{

    tokenList  *temp = malloc(sizeof(tokenList ));

    temp->name          = strdup(name);
    temp->type          = type;
    temp->returnType    = returnType;
    temp->size          = size;
    temp->line          = line;
    temp->scope         = (scope == PARENT_NONE ? NULL : strdup(scope));
    temp->parameter     = parameter;
    temp->next          = NULL;

    if(symPtr==NULL){
        symPtr=temp;
    }
    else{
        tokenList  *p = symPtr;
        while(p->next!=NULL)
                p=p->next;
        p->next=temp;
    }
    return temp;
}
```

```
void writeCode(FILE *icgOut)
{
    long icgLineCount  = 0;
    fprintf(icgOut, "\n\t\t\t  Intermediate Code Generated");
    fprintf(icgOut, "\n\t\t\t-------------------------------\n");

    if(codePtr==NULL){
        printf("\nNo code Generated\n");
    }
    else{
        threeAddressCode *p = codePtr;
        while(p)
        {
                if(p->gotoLine == -1){
                        fprintf(icgOut,"%-4lu %s\n",icgLineCount,p->code);
                }
                else{
                        fprintf(icgOut,"%-4lu %s %d\n",icgLineCount,p->code,p->gotoLine);
                }
                p = p->next;
                icgLineCount++;
        }
    }
}
void writeSymbolTable(FILE *symOut)
{
    fprintf(symOut, "\n\t\t\t   Symbol Table");
    fprintf(symOut, "\n\t\t\t-----------------\n\n");

    if(symPtr == NULL)
    {
        printf("\nSymbol table is empty");
    }
    else{
        fprintf(symOut,"   Name            DataType      ReturnType      Offset          Scope");
        fprintf(symOut,"\n------------------------------------------------------------------------\n");

        tokenList  *p= symPtr;
        while(p!=NULL)
        {
                fprintf(symOut,"%-20s %-14s %-14s %-12lu %-12s\n",

                      p->name,
                      dataTypeToString(p->type),
                      returnTypeToString(p->returnType),
```

```
                      p->size,
                      p->scope == NULL ? "None" : p->scope
                      );
                      p=p->next;
        }
    }
}
void addSymbolToQueue(char *name, tokenType type, unsigned long param_no) {
    tokenList  *temp = malloc(sizeof(tokenList ));
    temp->name          = strdup(name);
    temp->type          = type;
    switch(type){
                case INT_type: temp->size = 4;break;
                case FLOAT_type: temp->size = 8;break;
                case BOOL_type: temp->size = 1;break;
                default: temp->size = 4;
    }
    temp->parameter = param_no;
    temp->next=NULL;

    if (bufferPtr == NULL) {
        bufferPtr = temp;
    } else {
        tokenList  *p = bufferPtr;
        while (p->next!=NULL) {
                if(strcmp(p->name,name)==0){
                        printf("ERROR: %d : Multiple declaration\n",yylineno);
                        exit(1);
                }
            p=p->next;
          }
        p->next=temp;
    }
}
void clearQueue() {
    tokenList *temp;
    while(bufferPtr != NULL) {
        temp = bufferPtr;
        bufferPtr = bufferPtr->next;
        free(temp);
    }
}
void addFunctionPrototype(char *name, unsigned int paramCount, tokenReturnType   ret_type){

        if(strcmp(name, "main") == 0){
```

```
			printf("ERROR: %d : main func cannot declare as prototype\n",yylineno);
			exit(1);
		}
		tokenList  *temp = symPtr;
		while(temp!=NULL){
			if((strcmp(name,temp->name)==0)&&(temp->type==PROTO||temp->type==FUNC)){
				printf("ERROR: %d : Multiple function declaration of %s\n",yylineno,name);
				exit(1);
			}
			temp=temp->next;
		}
		appendToSymbolTable(name,PROTO,ret_type,0,0,NULL,paramCount);

		if(paramCount!=0 && bufferPtr==NULL){
			printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
			exit(1);
		}
		else{
			int cnt=1;
			if(bufferPtr!=NULL){
				tokenList *p=bufferPtr;
				while(p!=NULL){
					p->scope=name;
					p->parameter=cnt++;
					appendToSymbolTable(p->name,p->type,p->returnType,p->size,p->line,p->scope,p->parameter);
					p=p->next;

				}
				clearQueue();
			}

		}
}
tokenList * addSymbolToParameterQueue(tokenList * queue, char *name, tokenType type) {
    tokenList  *symbol = malloc(sizeof(tokenList ));
    symbol->name         = strdup(name);
    symbol->type         = type;
    symbol->next = false;
    if (queue == NULL) {
        return symbol;
    } else {
        tokenList  *entry = queue;
        while (entry->next) {
            entry = entry->next;
        }
```

```
        entry->next = symbol;
        return queue;
    }
}
void addFunction(char *name,unsigned int paramCount,tokenReturnType ret_type,int line) {
    tokenList  *symTable = symPtr,*prototype = NULL;
    unsigned int localOffset = 0;
    if(symTable != NULL){
			while(symTable){
				if( symTable->type == PROTO && 0 == strcasecmp(name,symTable->name)){
					prototype = symTable;
					break;
				}
				symTable = symTable->next;
			}
    }
    if(strcmp(name, "main") == 0){
		if(paramCount != 0){
			printf("ERROR: %d : main function should not have parameters\n",yylineno);
			exit(1);
		}
		else if(ret_type != Return_INT){
			printf("ERROR: %d : main function should return integer\n",yylineno);
			exit(1);
		}
		else{

			tokenList  *symbol = bufferPtr;
			int s = 0;
			while(symbol){
				int size = symbol->size;
				symbol->size = localOffset;
				localOffset+=size;
				globalOffset+=size;
				s+=size;

				symbol->scope = name;
				symbol->parameter = 0;
				appendToSymbolTable(symbol->name,symbol->type,symbol->returnType,symbol->size,symbol->line,symbol->scope,symbol->parameter);
				symbol=symbol->next;
			}
			tokenList  *newEntry= appendToSymbolTable(name,MAIN,ret_type,s,line,NULL,0);
			clearQueue();

		}
```

```
    }
    else if(!prototype){

        if(bufferPtr == NULL && paramCount != 0){
                printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
                exit(1);
        }
        else{
                int cnt = 1;
                int s = 0;
                tokenList  *symbol = bufferPtr;
                while(symbol!=NULL){
                        int size = symbol->size;
                        symbol->size = localOffset;
                        localOffset+=size;
                        globalOffset+=size;
                        s += size;

                        symbol->scope = name;
                        if(paramCount != 0){
                                symbol->parameter=cnt++;
                        }
                        else{
                                symbol->parameter=0;
                        }
                        appendToSymbolTable(symbol->name,symbol->type,symbol->returnType,symbol->size,symbol->line,symbol->scope,symbol->parameter);
                        symbol = symbol->next;
                }
            tokenList  *newEntry = appendToSymbolTable(name,FUNC,ret_type,s,line,NULL,paramCount);
            clearQueue();
        }
    }
    else{
        prototype->type = FUNC;
        prototype->line = line;
        if(bufferPtr == NULL && paramCount != 0){
                printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
                exit(1);
        }
        else{
                if(prototype->parameter != paramCount){
                        printf("ERROR: %d : Parameters are not matching with prototype %s\n",yylineno,name);
                        exit(1);
                }
                int s = 0;
                tokenList  *symbol = symPtr;
                for(int i = 0;i<paramCount;i++){
                        while(symbol->scope == NULL){
                                symbol=symbol->next;
                                if(strcmp(symbol->scope,name)==0&&symbol->scope != NULL){
                                        break;
                                }
                        }
                        if(symbol->type != bufferPtr->type){
                                printf("ERROR: %d : Parameter type mismatch in function %s\n",yylineno,name);
                                exit(1);
                        }
                        int size = symbol->size;
                        symbol->size = localOffset;
                        localOffset+=size;
                        globalOffset+=size;
                        s+=size;

                        tokenList  *temp = bufferPtr;
                        bufferPtr = bufferPtr->next;
                        free(temp);

                        symbol=symbol->next;
                }
                tokenList  *queuSymbol = bufferPtr;
                while(queuSymbol!=NULL){
                        int size = queuSymbol->size;
                        queuSymbol->size = localOffset;
                        localOffset+=size;
                        globalOffset+=size;
                        s+=size;

                        queuSymbol->scope = name;
                        queuSymbol->parameter = 0;
                        appendToSymbolTable(queuSymbol->name,queuSymbol->type,queuSymbol->returnType,queuSymbol->size,queuSymbol->line,
                                        queuSymbol->scope,queuSymbol->parameter);
                        queuSymbol = queuSymbol->next;
                }
                prototype->size = s;
                clearQueue();

        }
    }

}
```

```c
char* nextFloatVar(){
    char buffer[10];
    sprintf(buffer,"Tf_%d",++tempFloatCounter);
    return strdup(buffer);
}
char* nextIntVar(){
    char buffer[10];
    sprintf(buffer,"Ti_%d",++tempIntCounter);
    return strdup(buffer);
}
int nextquad(){
        return currentLine + 1;
}
tokenList * getSymbol(char *token){
    tokenList *p = symPtr;
    while(p!=NULL){
        if(p->name!=NULL && strcmp(token, p->name)==0){
            return p;
        }
        p=p->next;
    }
}
int getSymbolType(char *token) {
    tokenList  *p=bufferPtr;
    while(p!=NULL) {
        if(strcmp(token, p->name)==0)
                return(p->type);

        p=p->next;
    }
    return 0;
}
int getFunctionType(char *token){
        tokenList  *p=symPtr;
        while(p!=NULL){
                if(strcmp(token,p->name)==0 && (p->type==FUNC||p->type==PROTO)){
                        return p->returnType;
                }
        }

        return 0;
}
int checkAndGenerateParams(tokenList * queue, char* name ,int parameterCount){
        char buffer[50];
        tokenList  *cur = symPtr;
```

```c
        while(cur != NULL){
                if((cur->type == FUNC || cur->type == PROTO) && 0 == strcmp(name,cur->name)){
                        break;
                }
        }
        if(cur == NULL)
                return -1;
        int foundParams = 0;
        cur = symPtr;
        do{
                while(cur != NULL){
                        if(cur->scope != NULL && 0 == strcmp(name,cur->scope) && cur->parameter != 0){
                                break;
                        }
                        cur = cur->next;
                }
                if(cur == NULL || queue == NULL){
                        if(parameterCount == 0 && foundParams == 0)
                                return 0;
                        else
                                return -2;
                }
                else if(cur->type != queue->type){
                        return -3;
                }
                foundParams++;
                sprintf(buffer,"PARAM %s",queue->name);
                appendCode(buffer);
                cur = cur->next;
                tokenList  *temp = queue;
                queue = queue->next;
                free(temp);
        }while(foundParams != parameterCount);
        return 0;
}
```

icg.h

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <malloc.h>
#define PARENT_NONE NULL

typedef enum
{
     BOOL_type,
    INT_type,
    FLOAT_type,
     FUNC,
     MAIN,
     PROTO,
} tokenType;
typedef enum
{
    Return_VOID,
    Return_INT,
    Return_FLOAT

} tokenReturnType;
typedef enum
{
        CONST_type,
        VAR_type,
        NONE_type

} tokenConstType;

struct tokenList
{
    char *name;
    tokenType type;
    tokenReturnType returnType;
    long size;
    char *scope;
    long line;
    long parameter;

    struct tokenList *next;

};
typedef struct tokenList tokenList;

struct threeAddressCode
{
    char *code;
    int gotoLine;

    struct threeAddressCode *next;

};
typedef struct threeAddressCode threeAddressCode;

struct backPatchList
{
    threeAddressCode   *entry;
    struct backPatchList  *next;

};
typedef struct backPatchList   backPatchList ;

threeAddressCode* appendCode(char *code);
void backpatch(backPatchList * list, int gotoL);

backPatchList* mergelists(backPatchList  * a, backPatchList  * b);

backPatchList* appendToBackPatch(backPatchList  * list, threeAddressCode  * entry);

tokenList* appendToSymbolTable(char *name,tokenType type,tokenReturnType returnType,long size,long line,char *scope,long parameter);
void writeCode(FILE *icgOut);
void writeSymbolTable(FILE *symOut);

tokenList * addSymbolToParameterQueue(tokenList * queue, char *name, tokenType type);

int checkAndGenerateParams(tokenList * queue, char* name ,int parameterCount);

int getFunctionType(char *name);
int getSymbolType(char *name);
char* nextFloatVar();
char* nextIntVar();
char* nextBoolVar();
int nextquad();

void addFunction(char *name, unsigned int parameter_count, tokenReturnType  ret_type, int line);
tokenList * lookup(char *name);
extern tokenList *symbolTable;
```

EXECUTION OF THE CODE

The following is a shell script to automate the compilation and execution of the mini compiler consist of all phases.There is sample input source code given . Execution of the same is done and output is shown as sample output and respective symbol table is generated.

Shell Script

```sh
#!/bin/sh
lex lexicalAnalyzer.l
yacc -d syntaxChecker.y
gcc lex.yy.c y.tab.c icg.c -w -g
./a.out input.c
rm y.tab.c y.tab.h lex.yy.c
```

Sample Input

```c
#include <stdlib.h>
int main(){
	int a;
	a=40;
	if(a<50){
		a=60;
	}
	else{
		a=100;
	}
	return 1;
}

```

Symbol Table

symbolTable.txt (~/Projects/CD/Compiler-Design/Intermediate Code Generator/src) - gedit

```

                         Symbol Table
                      -----------------

   Name              DataType      ReturnType      Offset          Scope
-----------------------------------------------------------------------------
a                      Int            None            0            main
main                   Main           Int             4            None
```

Sample Three-address Code

```

                          Intermediate Code Generated
                      ------------------------------
0     GOTO 1
1     a := 40
2     IF (a < 50) GOTO 4
3     GOTO 6
4     a := 60
5     GOTO 7
6     a := 100
7     RETURN 1
```

TEST CASES

Different test cases are added to check implementation of ICG phase. Source code for the 6 test cases are added as screenshot below along with their three-address code.

Case1.c

```
//Testcase generate simple expresion statements
#include<stdio.h>
int main(){
	int a;
	float t;
	a=10;
	t=t*10;
	return 1;
}
```

```
                              Intermediate Code Generated
                            ------------------------------
0     GOTO 1
1     a := 10
2     Tf_1 := t * 10
3     t := Tf_1
4     RETURN 1
```

Case2.c

```
//testcase to generate control statments
#include<stdio.h>
int main(){
	int a;
	float t;
	a=20;
	if(a<30){
		t=3.0;
	}
	else{
		t=4.0;
	}
	return 2;
}
```

```
                          Intermediate Code Generated
                        -------------------------------
0    GOTO 1
1    a := 20
2    IF (a < 30) GOTO 4
3    GOTO 6
4    t := 3.0
5    GOTO 7
6    t := 4.0
7    RETURN 2
```

Case3.c

```c
//testcase to check looping statements
#include<stdio.h>
int main(){
        int i,s;
        s=0;
        for(i=0;i<10;i=i+2){
                s=s+i;
                while(s>30)
                {
                s=10;
                if(s<10)
                {

                                }
                                else
                                {
                                }
                }
        }
        return 2;
}
```

```
                          Intermediate Code Generated
                        -------------------------------
0    GOTO 1
1    s := 0
2    i := 0
3    IF (i < 10) GOTO 8
4    GOTO 18
5    Ti_1 := i + 2
6    i := Ti_1
7    GOTO 3
8    Ti_2 := s + i
9    s := Ti_2
10   IF (s > 30) GOTO 12
11   GOTO 5
12   s := 10
13   IF (s < 10) GOTO 15
14   GOTO 16
15   GOTO 10
16   GOTO 10
17   GOTO 5
18   RETURN 2
```

Case4.c

```c
//testcase to generate different functions code
#include<stdio.h>
int func1(){
        int s;
        s=56;
}
int func(int x,int y){
        x=x+y;
        func1();
}

int main(){
        int a;
        a=20;
        func1();
        return 1;
}
```

```

                              Intermediate Code Generated
                            -------------------------------
0    GOTO 3
1    s := 56
2    a := 20
3    RETURN 1
```

Case5.c

```c
//testcase to check main function errors
#include<stdio.h>
int main();
int main(){
        int y;
        y=20;
}
```

```

                              Intermediate Code Generated
                            -------------------------------
0    GOTO 1
1    y := 20
```

CONCLUSION

Intermediate Code Generation (ICG) is the fourth phase of a compiler. It is the final phase of the front end of a compiler. It generates intermediate code, that is a form which can be readily executed by machine. Intermediate code is converted to machine language using the last two phases which are platform dependent. We can take the intermediate code from the already existing compiler and build the last two parts. After this, the intermediate code is ready and it is then sent to the **Code Optimizer** phase which transforms the code so that it consumes fewer resources and produces more speed.