

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

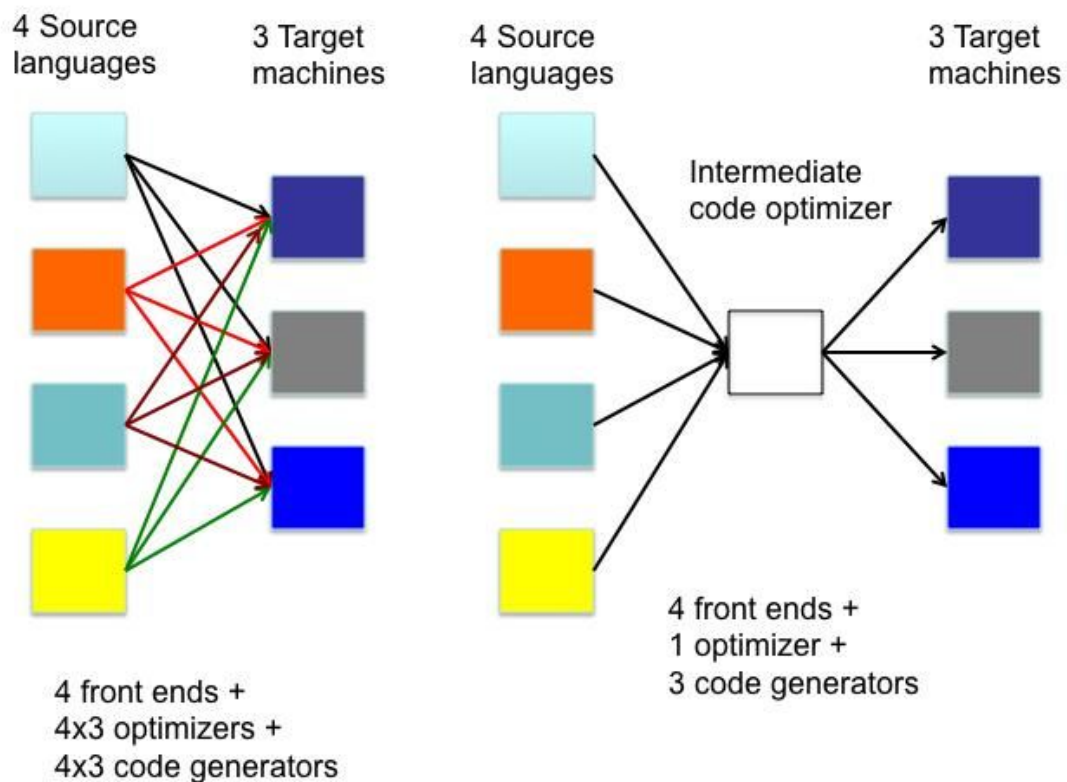
INTRODUCTION	3
INTERMEDIATE REPRESENTATION	5
THREE-ADDRESS CODE	6
IMPLEMENTATION	8
SOURCE CODE	9
lexicalAnalyzer.l	9
syntaxChecker.y	10
icg.c	19
icg.h	25
EXECUTION OF THE CODE	26
Shell Script	27
Sample Input	27
Symbol Table	27
Sample Three-address Code	28
TEST CASES	28
Case1.c	28
Case2.c	29
Case3.c	30
Case4.c	31
Case5.c	31
CONCLUSION	32

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 :

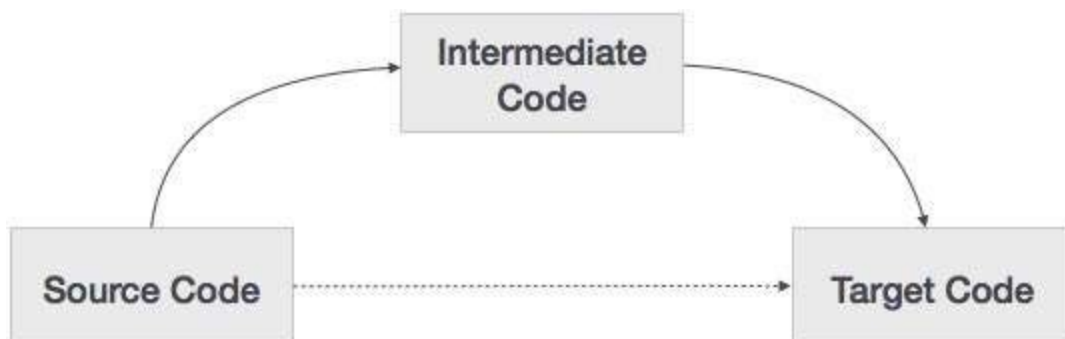
- 1) With **m** languages and **n** target machines, we need to write **m** front ends, **m × n** optimizers, and **m × n** code generators
- 2) The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused.



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 $x = y \text{ op } z$, 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 $a + b * c + d$:

- $T1 = b * c$
- $T2 = a + T1$
- $T3 = T2 + d$

where $T1, T2, T3$ 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* addToSymbolTable(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.

```
1 %option noyywrap
2 %option nomain
3 %option yylleno
4 %{
5 #include "lcg.h"
6 #include "y.tab.h"
7 int f=0;
8 %}
9 NEWLINE (\n|\n\\|\\r\\n)
10 WHITESPACE (" "|\t)
11 SINGLELINE_COMMENT ("/*".*{NEWLINE})
12 MULTILINE_COMMENT ("/*".*+*/")
13 PREPROCESSOR ("#"*. {NEWLINE})
14 INTEGER (0|[1-9][0-9]*)
15 FLOAT (0\.[0-9][0-9]*\.[0-9]+)
16 IDENTIFIER ([a-zA-Z_][a-zA-Z0-9_]*)
17 LIBFUNC "printf"|"scanf"
18 %%
19 ({NEWLINE}|{WHITESPACE})+ {}
20 ({SINGLELINE_COMMENT})|{MULTILINE_COMMENT}) {}
21 {PREPROCESSOR} {}
22 ^("+"|"-") {
23     if('+' == yytext[0]) {
24         return(U_PLUS);
25     } else {
26         return(U_MINUS);
27     }
28     f = 0;
29 }
30 (^("<"|">"|"<="|">="|"<<"|">>"|"&&"|"&&")) {f = 1;return(yytext[0]);}
31 (^("/"|"*"|"%"|"+")) {f = 1;return(yytext[0]);}
32 (^"+"|"-") {
33     if(f==1) {
34         if('+' == yytext[0]) {
35             return(U_PLUS);
36         }
37     } else {
38         return(U_MINUS);
39     }
40     f = 0;
41 } else {
42     return(yytext[0]);
43 }
44 }
45 }
```

```
46 "void" {return(VOID);}
47 "int" {return(INT);}
48 "float" {return(FLOAT);}
49 "constant" {return(CONSTANT);}
50 "if" {return(IF);}
51 "else" {return(ELSE);}
52 "return" {return(RETURN);}
53 "do" {return(DO);}
54 "while" {return(WHILE);}
55 "for" {return(FOR);}
56 "++" {return(INC_OP);}
57 "--" {return(DEC_OP);}
58 "=" {return(EQUAL);}
59 "!=" {return(NOT_EQUAL);}
60 ">=" {return(GREATER_OR_EQUAL);}
61 "<=" {return(LESS_OR_EQUAL);}
62 "<<" {return(SHIFTLEFT);}
63 "&&" {return(LOG_AND);}
64 "||" {return(LOG_OR);}
65 {INTEGER} {
66     yylval.expr.value = strdup(yytext);
67     yylval.expr.cType = CONST_type;
68     yylval.expr.type = INT_type;
69     f = 0;
70     return(CONSTANT);
71 }
72 {FLOAT} {
73     yylval.expr.value = strdup(yytext);
74     yylval.expr.cType = CONST_type;
75     yylval.expr.type = FLOAT_type;
76     f = 0;
77     return(CONSTANT);
78 }
79 {IDENTIFIER} {
80     yylval.str = malloc(strlen(yytext)*sizeof(char)+1);
81     strcpy(yylval.str, yytext);
82     f = 0;
83     return(IDENTIFIER);
84 }
85 }
86 %%
87
88
89
90 }
```

[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.

```
1 %{
2 #include "icg.h"
3 extern int yylineno;
4 extern char *yytext;
5 struct label{
6     char *funcName;|
7     int label;
8     struct label *next;
9 };
10 struct label *labelHead=NULL;
11
12 int label[10];
13 int funcCount=0;
14 int paramCount;
15 char icgQuad[50];
16 int funcLineNumber = 0;
17 %}
18 %token VOID INT FLOAT CONSTANT IDENTIFIER
19 %token IF ELSE RETURN DO WHILE FOR
20 %token INC_OP DEC_OP U_PLUS U_MINUS
21 %token EQUAL NOT_EQUAL GREATER_OR_EQUAL LESS_OR_EQUAL SHIFTLEFT LOG_AND LOG_OR
22
23 %right '='
24 %left LOG_OR
25 %left LOG_AND
26 %left '<' '>' LESS_OR_EQUAL GREATER_OR_EQUAL
27 %left EQUAL NOT_EQUAL
28 %left SHIFTLEFT
29 %left '+' '-'
30 %left '*' '/' '%'
31 %right U_PLUS U_MINUS '!'
32 %left INC_OP DEC_OP
33 %union
34 {
35     char
36     int
37     float
38     int
39     struct
40     {
41         char
42         int
43         int
44         struct BackpatchList*
45         struct BackpatchList*
46     } expr;
47     struct
48     {
49         struct BackpatchList*
50     } stmt;
51     struct
52     {
53         int
54         struct BackpatchList*
55     } mark;
56     struct
57     {
58         int
59         struct tokenList * queue;
60     } exp_list;
61 }
62
63 %type <str> id IDENTIFIER
64 %type <type> declaration var type
65 %type <expr> expression assignment CONSTANT
66 %type <stmt> statement statement_list matched_statement unmatched_statement program function_body function
67 %type <exp_list> exp_list
68 %type <mark> marker jump_marker
69 %start program_head
70 %%
71 program_head
72 : program
73 {
74     tokenList * mainFunc = getSymbol("main");
75     if(mainFunc == NULL){
76         printf("ERROR: Main function not found!\n");
77         yyerror();
78     }
79     backpatch($1.nextList,mainFunc->line+1);
80 }
81 ;
82 program
83 : jump_marker function
84 {
85     $$nextList = $1.nextList;
86     backpatch($2.nextList, nextquad());
87 }
88 | program function
89 {
90     $$nextList = $1.nextList;
```

```

101     backpatch($2.nextList, nextquad());
102 }
103 ;
104 function
105 : var_type id '(' parameter_list ')' ';'
106 {
107     if(labelHead==NULL){
108         struct label *temp=(struct label *)malloc(sizeof(label));
109         temp->name=strdup(id);
110         temp->label=funcCount;
111         temp->next=NULL;
112     }
113     else{
114         struct label *temp=(struct label *)malloc(sizeof(label));
115         temp->name=strdup(id);
116         temp->label=funcCount;
117         temp->next=NULL;
118         struct label *p=labelHead;
119         while(p->next!=NULL){
120             p=p->next;
121         }
122         p->next=temp;
123     }
124     funcCount++;
125     addFunctionPrototype($2, paramCount, $1);
126     paramCount = 0;
127     $$nextList = NULL;
128 }
129 | var_type id '(' parameter_list ')' function_body
130 {
131     addFunction($2, paramCount, $1, funcLineNumber);
132     paramCount = 0;
133     funcLineNumber = nextquad();
134     $$nextList = $6.nextList;
135 }
136 ;
137 function_body
138 : '{' statement_list '}'
139 {
140     $$nextList = $2.nextList;
141 }
142 | '{' declaration_list statement_list '}'
143 {

```

```

136     $$nextList = $3.nextList;
137 }
138 ;
139 declaration_list
140 : declaration ';'
141 | declaration_list declaration ';'
142 ;
143 declaration
144 : INT id
145 {
146     $$ = INT_type;
147     addSymbolToQueue($2, INT_type, 0);
148 }
149 | FLOAT id
150 {
151     $$ = FLOAT_type;
152     addSymbolToQueue($2, FLOAT_type, 0);
153 }
154 | declaration ',' id
155 {
156     if(INT_type == $1) {
157         addSymbolToQueue($3, INT_type, 0);
158     } else if(FLOAT_type == $1) {
159         addSymbolToQueue($3, FLOAT_type, 0);
160     }
161 }
162 ;
163 parameter_list
164 : INT id
165 {
166     paramCount++;
167     addSymbolToQueue($2, INT_type, paramCount);
168 }
169 | FLOAT id
170 {
171     paramCount++;
172     addSymbolToQueue($2, FLOAT_type, paramCount);
173 }
174 | parameter_list ',' INT id
175 {
176     paramCount++;
177     addSymbolToQueue($4, INT_type, paramCount);
178 }
179 | parameter_list ',' FLOAT id
180 {

```

```

181         paramCount++;|
182         addSymbolToQueue($4, FLOAT_type, paramCount);
183     }
184     | VOID
185     |
186     ;
187 var_type
188 : VOID
189 {
190     $$ = Return_VOID;
191 }
192 | INT
193 {
194     $$ = Return_INT;
195 }
196 | FLOAT
197 {
198     $$ = Return_FLOAT;
199 }
200 ;
201 statement_list
202 : statement
203 {
204     $$nextList = $1.nextList;
205 }
206 | statement_list marker statement
207 {
208     backpatch($1.nextList,$2.quad);
209     $$nextList = $3.nextList;
210 }
211 ;
212 statement
213 : matched_statement
214 {
215     $$nextList = $1.nextList;
216 }
217 | unmatched_statement
218 {
219     $$nextList = $1.nextList;
220 }
221 ;
222 matched_statement
223 : IF '(' assignment ')' marker matched_statement jump_marker ELSE marker matched_statement
224 {
225     backpatch($3.trueList,$5.quad);

```

```

226     backpatch($3.falseList,$9.quad);|
227     $$nextList = mergelists($7.nextList,$10.nextList);
228     $$nextList = mergelists($5.nextList,$6.nextList);
229 }
230 | assignment ';'
231 {
232     $$nextList = NULL;
233 }
234 | RETURN ';'
235 {
236     $$nextList = NULL;
237     sprintf(icgQuad,"RETURN");
238     appendCode(icgQuad);
239 }
240 | RETURN assignment ';'
241 {
242     $$nextList = NULL;
243     sprintf(icgQuad,"RETURN %s",$2.value);
244     appendCode(icgQuad);
245 }
246 | WHILE marker '(' assignment ')' marker matched_statement jump_marker
247 {
248     backpatch($4.trueList,$6.quad);
249     $$nextList = $4.falseList;
250     backpatch($7.nextList,$2.quad);
251     backpatch($8.nextList,$2.quad);
252 }
253 | DO marker statement WHILE '(' marker assignment ')' ';'
254 {
255     backpatch($3.nextList,$6.quad);
256     backpatch($7.trueList,$2.quad);
257     $$nextList = $7.falseList;
258 }
259 | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker matched_statement jump_marker
260 {
261     if(BOOL_type == $3.type || BOOL_type == $9.type) {
262         printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
263         yyerror();
264     }
265     if(BOOL_type != $6.type) {
266         printf("error, 2nd argument of for loop must be boolean\n");
267         yyerror();
268     }
269     backpatch($3.trueList, $5.quad);
270     backpatch($13.nextList, $8.quad);

```

```

271     backpatch($14.nextList, $8.quad);
272     $$nextList = $6.falseList;
273     backpatch($6.trueList, $12.quad);
274     backpatch($9.trueList, $5.quad);
275     backpatch($10.nextList, $5.quad);
276
277     }
278     | '{' statement_list '}'
279     {
280         $$nextList = $2.nextList;
281     }
282     | '{' '}'
283     {
284         $$nextList = NULL;
285     }
286 ;
287 unmatched_statement
288 : IF '(' assignment ')' marker statement
289 {
290     backpatch($3.trueList,$5.quad);
291     $$nextList = mergelists($3.falseList,$6.nextList);
292 }
293 | WHILE marker '(' assignment ')' marker unmatched_statement jump_marker
294 {
295     backpatch($4.trueList,$6.quad);
296     $$nextList = $4.falseList;
297     backpatch($7.nextList,$2.quad);
298     backpatch($8.nextList,$2.quad);
299 }
300 | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker unmatched_statement jump_marker
301 {
302     if(BOOL_type == $3.type || BOOL_type == $9.type) {
303         printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
304         yyerror();
305     }
306     if(BOOL_type != $6.type) {
307         printf("error, 2nd argument of for loop must be boolean\n");
308         yyerror();
309     }
310     backpatch($3.trueList, $5.quad);
311     backpatch($13.nextList, $8.quad);
312     backpatch($14.nextList, $8.quad);
313     $$nextList = $6.falseList;
314     backpatch($6.trueList, $12.quad);
315     backpatch($9.trueList, $5.quad);

```

```

316     backpatch($10.nextList, $5.quad);
317 }
318 | IF '(' assignment ')' marker matched_statement jump_marker ELSE marker unmatched_statement
319 {
320     backpatch($3.trueList,$5.quad);
321     backpatch($3.falseList,$9.quad);
322     $$nextList = mergelists($7.nextList,$10.nextList);
323     $$nextList = mergelists($$.nextList,$6.nextList);
324 }
325 ;
326 assignment
327 : expression
328 {
329     $$=$1;
330 }
331 | id '=' expression
332 {
333     int destType = getSymbolType($1);
334     if(destType == 0){
335         printf("ERROR: Not in scope");
336     }
337     if(destType != $3.type) {
338         printf("Type error on line: %d\n", yylineno);
339         yyerror();
340     }
341     sprintf(lcgQuad,"%s := %s",$1,$3.value);
342     appendCode(lcgQuad);
343     $$type = destType;
344     $$trueList = $3.trueList;
345     $$cType = VAR_type;
346     $$value = $1;
347 }
348 ;
349 expression
350 : INC_OP expression
351 {
352     if($2.type != INT_type){
353         printf("ERROR: Increment not allowed for types different than Integer.\n");
354         yyerror();
355     }
356     //Create a variable if needed
357     if($2.cType != VAR_type){
358         char *var = nextIntVar();
359         sprintf(lcgQuad,"%s += %s",var,$2.value);
360     }

```



```

361         appendCode(lcgQuad);
362         free($2.value);
363         $2.value = var;
364         $2.type = INT_type;
365         $2.cType = VAR_type;
366     }
367     sprintf(lcgQuad, "%s := %s + 1", $2.value, $2.value);
368     appendCode(lcgQuad);
369     //Set the attributes
370     $$ = $2;
371     $$trueList = NULL;
372     $$falseList = NULL;
373 }
374 | DEC_OP expression
375 {
376     if($2.type != INT_type){
377         printf("ERROR: Decrement not allowed for types different than Integer.\n");
378         yyerror();
379     }
380     //Create a variable if needed
381     if($2.cType != VAR_type){
382         char *var = nextIntVar();
383         sprintf(lcgQuad, "%s := %s", var, $2.value);
384         appendCode(lcgQuad);
385         free($2.value);
386         $2.value = var;
387         $2.type = INT_type;
388         $2.cType = VAR_type;
389     }
390     sprintf(lcgQuad, "%s := %s - 1", $2.value, $2.value);
391     appendCode(lcgQuad);
392     //Set the attributes
393     $$ = $2;
394     $$trueList = NULL;
395     $$falseList = NULL;
396 }
397 | expression LOG_OR marker expression
398 {
399     if(BOOL_type != $1.type) {
400         sprintf(lcgQuad, "IF (%s <> 0) GOTO", $1.value);
401         $1.trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
402         sprintf(lcgQuad, "GOTO");
403         $1.falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
404     }
405     if(BOOL_type != $4.type) {

```

```

406         sprintf(lcgQuad, "IF (%s <> 0) GOTO", $4.value);
407         $4.trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
408         sprintf(lcgQuad, "GOTO");
409         $4.falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
410     }
411     $$trueList = mergeLists($1.trueList, $4.trueList);
412     backpatch($1.falseList, $3.quad);
413     $$falseList = $4.falseList;
414     $$type = BOOL_type;
415 }
416 | expression LOG_AND marker expression
417 {
418     if(BOOL_type != $1.type) {
419         sprintf(lcgQuad, "IF (%s <> 0) GOTO", $1.value);
420         $1.trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
421         sprintf(lcgQuad, "GOTO");
422         $1.falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
423     }
424     if(BOOL_type != $4.type) {
425         sprintf(lcgQuad, "IF (%s <> 0) GOTO", $4.value);
426         $4.trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
427         sprintf(lcgQuad, "GOTO");
428         $4.falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
429     }
430     $$falseList = mergeLists($1.falseList, $4.falseList);
431     backpatch($1.trueList, $3.quad);
432     $$trueList = $4.trueList;
433     $$type = BOOL_type;
434 }
435 | expression NOT_EQUAL expression
436 {
437     if($1.type != INT_type && $1.type != FLOAT_type){
438         printf("ERROR: Only Integer, Float and Bool values allowed in comparisons.\n");
439         yyerror();
440     }
441     sprintf(lcgQuad, "IF (%s <> %s) GOTO", $1.value, $3.value);
442     $$trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
443     sprintf(lcgQuad, "GOTO");
444     $$falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
445     $$value = "TrueFalse Only!";
446     $$type = BOOL_type;
447     $$cType = NONE_type;
448 }
449
450

```

```

451 | expression EQUAL expression
452 {
453     if($1.type != INT_type && $1.type != FLOAT_type){
454         printf("ERROR: Only Integer, Float and Bool values allowed in comparsons.\n");
455         yyerror();
456     }
457     sprintf(icgQuad,"IF (%s = %s) GOTO",$1.value,$3.value);
458     $$trueList = appendToBackPatch(NULL, appendCode(icgQuad));
459     sprintf(icgQuad,"GOTO");
460     $$falseList = appendToBackPatch(NULL, appendCode(icgQuad));
461     if(BOOL_type == $1.type) {
462         $$trueList = mergelists($$trueList, $1.trueList);
463         $$falseList = mergelists($$falseList, $1.falseList);
464     }
465     if(BOOL_type == $3.type) {
466         $$trueList = mergelists($$trueList, $3.trueList);
467         $$falseList = mergelists($$falseList, $3.falseList);
468     }
469     $$value = "TrueFalse Only!";
470     $$type = BOOL_type;
471     $$cType = NONE_type;
472 }
473 | expression GREATER_OR_EQUAL expression
474 {
475     if($1.type != INT_type && $1.type != FLOAT_type){
476         printf("ERROR: Only Integer, Float and Bool values allowed in comparsons.\n");
477         yyerror();
478     }
479     sprintf(icgQuad,"IF (%s >= %s) GOTO",$1.value,$3.value);
480     $$trueList = appendToBackPatch(NULL, appendCode(icgQuad));
481     sprintf(icgQuad,"GOTO");
482     $$falseList = appendToBackPatch(NULL, appendCode(icgQuad));
483     $$value = "TrueFalse Only!";
484     $$type = BOOL_type;
485     $$cType = NONE_type;
486 }
487 | expression LESS_OR_EQUAL expression
488 {
489     if($1.type != INT_type && $1.type != FLOAT_type){
490         printf("ERROR: Only Integer, Float and Bool values allowed in comparsons.\n");
491         yyerror();
492     }
493     sprintf(icgQuad,"IF (%s <= %s) GOTO",$1.value,$3.value);
494     $$trueList = appendToBackPatch(NULL, appendCode(icgQuad));
495     sprintf(icgQuad,"GOTO");

```

```

496     $$falseList = appendToBackPatch(NULL, appendCode(icgQuad));
497     $$value = "TrueFalse Only!";
498     $$type = BOOL_type;
499     $$cType = NONE_type;
500 }
501 | expression '>' expression
502 {
503     if($1.type != INT_type && $1.type != FLOAT_type){
504         printf("ERROR: Only Integer, Float and Bool values allowed in comparsons.\n");
505         yyerror();
506     }
507     sprintf(icgQuad,"IF (%s > %s) GOTO",$1.value,$3.value);
508     $$trueList = appendToBackPatch(NULL, appendCode(icgQuad));
509     sprintf(icgQuad,"GOTO");
510     $$falseList = appendToBackPatch(NULL, appendCode(icgQuad));
511     $$value = "TrueFalse Only!";
512     $$type = BOOL_type;
513     $$cType = NONE_type;
514 }
515 | expression '<' expression
516 {
517     if($1.type != INT_type && $1.type != FLOAT_type){
518         printf("ERROR: Only Integer, Float and Bool values allowed in comparsons.\n");
519         yyerror();
520     }
521     sprintf(icgQuad,"IF (%s < %s) GOTO",$1.value,$3.value);
522     $$trueList = appendToBackPatch(NULL, appendCode(icgQuad));
523     sprintf(icgQuad,"GOTO");
524     $$falseList = appendToBackPatch(NULL, appendCode(icgQuad));
525     $$value = "TrueFalse Only!";
526     $$type = BOOL_type;
527     $$cType = NONE_type;
528 }
529 | expression '+' expression
530 {
531     if($1.type != INT_type && $1.type != FLOAT_type && $3.type != INT_type && $3.type != FLOAT_type){
532         printf("ERROR: Only integer and float values allowed when adding numbers.\n");
533         yyerror();
534     }
535     int type = 0;
536     if($1.type == $3.type){
537         type = $1.type;
538     }
539     else{
540         type = FLOAT_type;

```

```

541
542
543     char* var = NULL;
544     switch(type){
545         case INT_type: var = nextIntVar();break;
546         case FLOAT_type:var = nextFloatVar();break;
547     }
548     char buffer[50];
549     sprintf(icgQuad,"%s := %s + %s",var,$1.value,$3.value);
550     appendCode(icgQuad);
551     $$value = var;
552     $$type = type;
553     $$cType = VAR_type;
554     $$trueList = NULL;
555     $$falseList = NULL;
556 }
557 | expression '-' expression
558 {
559     if($1.type != INT_type && $1.type!= FLOAT_type && $3.type != INT_type && $3.type != FLOAT_type){
560         printf("ERROR: Only integer and float values allowed when subtracting numbers.\n");
561         yyerror();
562     }
563     int type = 0;
564     if($1.type == $3.type){
565         type = $1.type;
566     }
567     else{
568         type = FLOAT_type;
569     }
570
571     char* var = NULL;
572     switch(type){
573         case INT_type: var = nextIntVar();break;
574         case FLOAT_type:var = nextFloatVar();break;
575     }
576     char buffer[50];
577     sprintf(icgQuad,"%s := %s - %s",var,$1.value,$3.value);
578     appendCode(icgQuad);
579     $$value = var;
580     $$type = type;
581     $$cType = VAR_type;
582     $$trueList = NULL;
583     $$falseList = NULL;
584 }
585 | expression '*' expression

```

```

586
587     if($1.type != INT_type && $1.type!= FLOAT_type && $3.type != INT_type && $3.type != FLOAT_type){
588         printf("ERROR: Only integer and float values allowed when multiplying numbers.\n");
589         yyerror();
590     }
591     int type = 0;
592     if($1.type == $3.type){
593         type = $1.type;
594     }
595     else{
596         type = FLOAT_type;
597     }
598
599     char* var = NULL;
600     switch(type){
601         case INT_type: var = nextIntVar();break;
602         case FLOAT_type:var = nextFloatVar();break;
603     }
604     char buffer[50];
605     sprintf(icgQuad,"%s := %s * %s",var,$1.value,$3.value);
606     appendCode(icgQuad);
607     $$value = var;
608     $$type = type;
609     $$cType = VAR_type;
610     $$trueList = NULL;
611     $$falseList = NULL;
612 | expression '/' expression
613 {
614     if($1.type != INT_type && $1.type!= FLOAT_type && $3.type != INT_type && $3.type != FLOAT_type){
615         printf("ERROR: Only integer and float values allowed when dividing numbers.\n");
616         yyerror();
617     }
618     int type = 0;
619     if($1.type == $3.type){
620         type = $1.type;
621     }
622     else{
623         type = FLOAT_type;
624     }
625
626     char* var = NULL;
627     switch(type){
628         case INT_type: var = nextIntVar();break;
629         case FLOAT_type:var = nextFloatVar();break;
630     }
631     char buffer[50];

```



```

631     sprintf(lcgQuad, "%s := %s / %s", var, $1.value, $3.value);
632     appendCode(lcgQuad);
633     $$value = var;
634     $$type = type;
635     $$cType = VAR_type;
636     $$trueList = NULL;
637     $$falseList = NULL;
638 }
639 | expression '%' expression
640 {
641     if($1.type != INT_type && $1.type != FLOAT_type && $3.type != INT_type && $3.type != FLOAT_type){
642         printf("ERROR: Only integer and float values allowed when calculating mod.\n");
643         yyerror();
644     }
645     int type = 0;
646     if($1.type == $3.type){
647         type = $1.type;
648     }
649     else{
650         type = FLOAT_type;
651     }
652     char* var = NULL;
653     switch(type){
654         case INT_type: var = nextIntVar();break;
655         case FLOAT_type: var = nextFloatVar();break;
656     }
657     char buffer[50];
658     sprintf(lcgQuad, "%s := %s %% %s", var, $1.value, $3.value);
659     appendCode(lcgQuad);
660     $$value = var;
661     $$type = type;
662     $$cType = VAR_type;
663     $$trueList = NULL;
664     $$falseList = NULL;
665 }
666 | '!' expression
667 {
668     if($2.type != BOOL_type){
669         if($2.type != INT_type && $2.type != FLOAT_type){
670             printf("ERROR: Only Bool, Int and Float allowed in logical expressions!\n");
671             yyerror();
672         }
673         sprintf(lcgQuad, "IF (%s <> 0) GOTO", $2.value);
674         $$falseList = appendToBackPatch(NULL, appendCode(lcgQuad));
675         sprintf(lcgQuad, "GOTO", $2.value);

```

```

676         $$trueList = appendToBackPatch(NULL, appendCode(lcgQuad));
677     }
678     else{
679         $$ = $2;
680         $$trueList = $2.falseList;
681         $$falseList = $2.trueList;
682     }
683 }
684 | U_PLUS expression
685 {
686     if(INT_type != $2.type && FLOAT_type != $2.type) {
687         yyerror();
688     }
689     $$ = $2;
690 }
691 | U_MINUS expression
692 {
693     $$ = $2;
694     if(INT_type == $2.type) {
695         $$value = nextIntVar();
696     } else if (FLOAT_type == $2.type) {
697         $$value = nextFloatVar();
698     } else {
699         yyerror();
700     }
701     sprintf(lcgQuad, "%s := -%s", $$value, $2.value);
702     appendCode(lcgQuad);
703 }
704 | CONSTANT
705 {
706     $$value = strdup(yytext);
707     $$trueList = NULL;
708     $$falseList = NULL;
709 }
710 | '(' expression ')'
711 {
712     $$ = $2;
713 }
714 | id '(' exp_list ')'
715 {
716     int varType = getFunctionType($1);
717     if(varType == 0){
718         printf("ERROR: Function %s not defined!\n", $1);
719         yyerror();
720     }

```

```

721     char* var = NULL;
722     switch(varType){
723     case Return_INT:
724         var = nextIntVar();
725         $$type = INT_type;
726         break;
727     case Return_FLOAT:
728         var = nextFloatVar();
729         $$type = FLOAT_type;
730         break;
731     }
732     $$value = var;
733     $$cType = VAR_type;
734     checkAndGenerateParams($$.queue,$1,$$.count);
735     sprintf(lcgQuad,"%s := CALL %s, %d",var,$1,$$.count);
736     appendCode (lcgQuad);
737 }
738 | id '(' ' '
739 {
740     int varType = getFunctionType($1);
741     if(varType == 0){
742         printf("ERROR: Function %s not defined!\n",$1);
743         yyerror();
744     }
745     char* var = NULL;
746     switch(varType){
747     case Return_INT:
748         var = nextIntVar();
749         $$type = INT_type;
750         break;
751     case Return_FLOAT:
752         var = nextFloatVar();
753         $$type = FLOAT_type;
754         break;
755     }
756     $$value = var;
757     $$cType = VAR_type;
758     checkAndGenerateParams(NULL,$1,0);
759     sprintf(lcgQuad,"%s := CALL %s, %d",var,$1,0);
760     appendCode (lcgQuad);
761 }
762 | id {
763     int varType = getSymbolType($1);

```

```

766     if(varType == 0){
767         printf("ERROR: Variable %s not in scope!\n",$1);
768         yyerror();
769     }
770     $$value = $1;
771     $$type = varType;
772     $$cType = VAR_type;
773 }
774 ;
775 exp_list
776 : expression
777 {
778     if($1.type != INT_type && $1.type != FLOAT_type){
779         printf("ERROR: Only Integer and Float are allowed as parameter types.\n");
780         yyerror();
781     }
782     $$queue = addSymbolToParameterQueue(NULL,$1.value,$1.type);
783     $$count = 1;
784 }
785 | exp_list ' ' expression
786 {
787     if($3.type != INT_type && $3.type != FLOAT_type){
788         printf("ERROR: Only Integer and Float are allowed as parameter types.\n");
789         yyerror();
790     }
791     $$queue = addSymbolToParameterQueue($1.queue,$3.value,$3.type);
792     $$count = $1.count + 1;
793 }
794 ;
795 id | IDENTIFIER
796 {
797     $$ = strdup(yytext);
798 }
799 ;
800 marker
801 : {
802     $$quad = nextquad();
803 };
804 jump_marker
805 : {
806     $$quad = nextquad();
807     sprintf(lcgQuad,"%GOTO");
808     $$nextList = appendToBackPatch(NULL, appendCode (lcgQuad));
809 };
810 %%

```

icg.c

```

1 #include "icg.h"
2 #include "y.tab.h"
3
4 tokenList *synPtr = NULL;
5 tokenList *bufferPtr = NULL;
6 threeAddressCode *codePtr = NULL;
7
8 extern int yylineno;
9 extern int yyin;
10
11 int globalOffset = 0, tempIntCounter = 0, tempFloatCounter = 0, currentLine = -1;
12
13 void yyerror() {
14     printf("ERROR: %d : Syntactical Error\n", yylineno);
15     exit(1);
16 }
17
18 int main(int argc, char **argv) {
19     yyin = fopen(argv[1], "r");
20     yyparse();
21     printf("\n\n\t Parsing Completed\n");
22     printf("Three Address Code and Symbol table are generated\n\n");
23     FILE *icgPtr = fopen("threeAddressCode.txt", "w");
24     writeCode(icgPtr);
25     FILE *synPtr = fopen("symbolTable.txt", "w");
26     writeSymbolTable(synPtr);
27     fclose(icgPtr);
28     fclose(synPtr);
29     printf("\n");
30     return(0);
31 }
32
33 static const char* returnTypeToString(tokenReturnType type)
34 {
35     switch(type)
36     {
37         case Return_VOID:
38             return "None";
39
40         case Return_INT:
41             return "Int";
42
43         case Return_FLOAT:
44             return "Real";
45     }

```

```

46 }
47
48 static const char* dataTypeToString(tokenType type)
49 {
50     switch(type)
51     {
52
53         case INT_type:
54             return "Int";
55
56         case FLOAT_type:
57             return "Real";
58
59         case BOOL_type:
60             return "Bool";
61
62         case FUNC:
63             return "Func";
64         case PROTO:
65             return "Proto";
66
67         case MAIN:
68             return "Main";
69     }
70 }
71
72 threeAddressCode* appendCode(char *code){
73     threeAddressCode *temp = malloc(sizeof(threeAddressCode ));
74
75     temp->code = (char *) malloc(sizeof(char)*strlen(code));
76     strcpy(temp->code, code);
77     temp->gotoLine = -1;
78     temp->next = NULL;
79
80
81     if(codePtr == NULL){
82         codePtr = temp;
83     }
84     else{
85         threeAddressCode *p = codePtr;
86         while(p->next != NULL){
87             p = p->next;
88         }
89         p->next = temp;
90     }

```

```

91     }
92     currentLine++;
93     return temp;
94 }
95
96 void backpatch(backPatchList* list, int gotoL){
97     if(list == NULL){
98         return;
99     } else{
100         backPatchList* temp;
101         while(list){
102             if(list->entry != NULL){
103                 list->entry->gotoLine = gotoL;
104             }
105             //printf("backpatching: %s",list->entry->code);
106             temp = list;
107             list = list->next;
108             free(temp);
109         }
110     }
111 }
112
113 backPatchList* mergelists(backPatchList* a, backPatchList* b){
114     if(a != NULL && b == NULL){
115         return a;
116     }
117     else if(a == NULL && b != NULL){
118         return b;
119     }
120     else if(a == NULL && b == NULL){
121         return NULL;
122     }
123     else{
124         backPatchList* temp = a;
125         while(a->next){
126             a = a->next;
127         }
128         a->next = b;
129         return temp;
130     }
131 }
132
133 backPatchList* appendToBackPatch(backPatchList* p, threeAddressCode* newCode){
134
135     if(newCode == NULL){

```

```

136         return p;
137     }
138     else if(p == NULL){
139         backPatchList* temp = malloc(sizeof(backPatchList));
140         temp->entry = newCode;
141         temp->next = NULL;
142         return temp;
143     }
144     else{
145         backPatchList* temp = malloc(sizeof(backPatchList));
146         temp->entry = newCode;
147         temp->next=NULL;
148         while(p->next!=NULL){
149             p=p->next;
150         }
151         p->next = temp;
152         return temp;
153     }
154 }
155
156 tokenList* appendToSymbolTable(char *name,tokenType type,tokenReturnType returnType,long size,long line,char *scope,long parameter)
157 {
158     tokenList *temp = malloc(sizeof(tokenList ));
159
160     temp->name      = strdup(name);
161     temp->type      = type;
162     temp->returnType = returnType;
163     temp->size      = size;
164     temp->line      = line;
165     temp->scope     = (scope == PARENT_NONE ? NULL : strdup(scope));
166     temp->parameter = parameter;
167     temp->next      = NULL;
168
169     if(symPtr==NULL){
170         symPtr=temp;
171     }
172     else{
173         tokenList *p = symPtr;
174         while(p->next!=NULL)
175             p=p->next;
176         p->next=temp;
177     }
178     return temp;
179 }
180

```



```

371         printf("ERROR: %d : main func cannot declare as prototype\n",yylineno);
372         exit(1);
373     }
374     tokenList *temp = symPtr;
375     while(temp!=NULL){
376         if((strcmp(name,temp->name)==0)&&(temp->type==PROTO||temp->type==FUNC)){
377             printf("ERROR: %d : Multiple function declaration of %s\n",yylineno,name);
378             exit(1);
379         }
380         temp=temp->next;
381     }
382     addToSymbolTable(name,PROTO,ret_type,0,0,NULL,paramCount);
383
384     if(paramCount!=0 && bufferPtr==NULL){
385         printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
386         exit(1);
387     }
388     else{
389         int cnt=1;
390         if(bufferPtr!=NULL){
391             tokenList *p=bufferPtr;
392             while(p!=NULL){
393                 p->scope=name;
394                 p->parameter=cnt++;
395                 addToSymbolTable(p->name,p->type,p->returnType,p->size,p->line,p->scope,p->parameter);
396                 p=p->next;
397             }
398             clearQueue();
399         }
400     }
401 }
402
403 tokenList * addSymbolToParameterQueue(tokenList * queue, char *name, tokenType type) {
404     tokenList *symbol = malloc(sizeof(tokenList ));
405     symbol->name = strdup(name);
406     symbol->type = type;
407     symbol->next = false;
408     if (queue == NULL) {
409         return symbol;
410     } else {
411         tokenList *entry = queue;
412         while (entry->next)
413             entry = entry->next;
414         entry->next = symbol;
415     }

```

```

416     entry->next = symbol;
417     return queue;
418 }
419
420 void addFunction(char *name,unsigned int paramCount,tokenReturnType ret_type,int line) {
421     tokenList *symTable = symPtr,*prototype = NULL;
422     unsigned int localOffset = 0;
423     if(symTable != NULL){
424         while(symTable){
425             if( symTable->type == PROTO && 0 == strcmp(name,symTable->name)){
426                 prototype = symTable;
427                 break;
428             }
429             symTable = symTable->next;
430         }
431     }
432     if(strcmp(name, "main") == 0){
433         if(paramCount != 0){
434             printf("ERROR: %d : main function should not have parameters\n",yylineno);
435             exit(1);
436         }
437         else if(ret_type != Return_INT){
438             printf("ERROR: %d : main function should return integer\n",yylineno);
439             exit(1);
440         }
441         else{
442             tokenList *symbol = bufferPtr;
443             int s = 0;
444             while(symbol){
445                 int size = symbol->size;
446                 symbol->size = localOffset;
447                 localOffset+=size;
448                 globalOffset+=size;
449                 s+=size;
450             }
451             symbol->scope = name;
452             symbol->parameter = 0;
453             addToSymbolTable(symbol->name,symbol->type,symbol->returnType,symbol->size,symbol->line,symbol->scope,symbol->parameter);
454             symbol=symbol->next;
455         }
456         tokenList *newEntry= addToSymbolTable(name,MAIN,ret_type,s,line,NULL,0);
457         clearQueue();
458     }
459 }
460

```

```

361 }
362 else if(!prototype){
363
364     if(bufferPtr == NULL && paramCount != 0){
365         printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
366         exit(1);
367     }
368     else{
369         int cnt = 1;
370         int s = 0;
371         tokenList *symbol = bufferPtr;
372         while(symbol!=NULL){
373             int size = symbol->size;
374             symbol->size = localOffset;
375             localOffset+=size;
376             globalOffset+=size;
377             s += size;
378
379             symbol->scope = name;
380             if(paramCount != 0){
381                 symbol->parameter=cnt++;
382             }
383             else{
384                 symbol->parameter=0;
385             }
386             appendToSymbolTable(symbol->name,symbol->type,symbol->returnType,symbol->size,symbol->line,symbol->scope,symbol->parameter);
387             symbol = symbol->next;
388         }
389         tokenList *newEntry = appendToSymbolTable(name,FUNC,ret_type,s,line,NULL,paramCount);
390         clearQueue();
391     }
392 }
393 else{
394     prototype->type = FUNC;
395     prototype->line = line;
396     if(bufferPtr == NULL && paramCount != 0){
397         printf("ERROR: %d : Parameter mismatch of function %s\n",yylineno,name);
398         exit(1);
399     }
400     else{
401         if(prototype->parameter != paramCount){
402             printf("ERROR: %d : Parameters are not matching with prototype %s\n",yylineno,name);
403             exit(1);
404         }
405         int s = 0;

```

```

406         tokenList *symbol = symPtr;
407         for(int i = 0;i<paramCount;i++){
408             while(symbol->scope == NULL){
409                 symbol=symbol->next;
410                 if(strcmp(symbol->scope,name)==0&&symbol->scope != NULL){
411                     break;
412                 }
413             }
414             if(symbol->type != bufferPtr->type){
415                 printf("ERROR: %d : Parameter type mismatch in function %s\n",yylineno,name);
416                 exit(1);
417             }
418             int size = symbol->size;
419             symbol->size = localOffset;
420             localOffset+=size;
421             globalOffset+=size;
422             s+=size;
423
424             tokenList *temp = bufferPtr;
425             bufferPtr = bufferPtr->next;
426             free(temp);
427
428             symbol=symbol->next;
429         }
430         tokenList *queueSymbol = bufferPtr;
431         while(queueSymbol!=NULL){
432             int size = queueSymbol->size;
433             queueSymbol->size = localOffset;
434             localOffset+=size;
435             globalOffset+=size;
436             s+=size;
437
438             queueSymbol->scope = name;
439             queueSymbol->parameter = 0;
440             appendToSymbolTable(queueSymbol->name,queueSymbol->type,queueSymbol->returnType,queueSymbol->size,queueSymbol->line,
441                 queueSymbol->scope,queueSymbol->parameter);
442             queueSymbol = queueSymbol->next;
443         }
444         prototype->size = s;
445         clearQueue();
446     }
447 }
448 }
449
450

```

```

451 char* nextFloatVar(){
452     char buffer[10];
453     sprintf(buffer,"Tf %d",++tempFloatCounter);
454     return strdup(buffer);
455 }
456 char* nextIntVar(){
457     char buffer[10];
458     sprintf(buffer,"Ti %d",++tempIntCounter);
459     return strdup(buffer);
460 }
461 int nextquad(){
462     return currentLine + 1;
463 }
464 tokenList * getSymbol(char *token){
465     tokenList *p = symPtr;
466     while(p!=NULL){
467         if(p->name!=NULL && strcmp(token, p->name)==0){
468             return p;
469         }
470         p=p->next;
471     }
472 }
473 int getSymbolType(char *token) {
474     tokenList *p=bufferPtr;
475     while(p!=NULL) {
476         if(strcmp(token, p->name)==0)
477             return(p->type);
478         p=p->next;
479     }
480     return 0;
481 }
482 }
483 int getFunctionType(char *token){
484     tokenList *p=symPtr;
485     while(p!=NULL){
486         if(strcmp(token,p->name)==0 && (p->type==FUNC||p->type==PROTO)){
487             return p->returnType;
488         }
489     }
490     return 0;
491 }
492 }
493 int checkAndGenerateParams(tokenList * queue, char* name ,int parameterCount){
494     char buffer[50];
495     tokenList *cur = symPtr;

```

```

496     while(cur != NULL){
497         if((cur->type == FUNC || cur->type == PROTO) && 0 == strcmp(name,cur->name)){
498             break;
499         }
500     }
501     if(cur == NULL)
502         return -1;
503     int foundParams = 0;
504     cur = symPtr;
505     do{
506         while(cur != NULL){
507             if(cur->scope != NULL && 0 == strcmp(name,cur->scope) && cur->parameter != 0){
508                 break;
509             }
510             cur = cur->next;
511         }
512         if(cur == NULL || queue == NULL){
513             if(parameterCount == 0 && foundParams == 0)
514                 return 0;
515             else
516                 return -2;
517         }
518         else if(cur->type != queue->type){
519             return -3;
520         }
521         foundParams++;
522         sprintf(buffer,"PARAM %s",queue->name);
523         appendCode(buffer);
524         cur = cur->next;
525         tokenList *temp = queue;
526         queue = queue->next;
527         free(temp);
528     }while(foundParams != parameterCount);
529     return 0;
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }

```

icg.h


```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <stdbool.h>
5 #include <malloc.h>
6 #define PARENT_NONE NULL
7
8 typedef enum
9 {
10     BOOL_type,
11     INT_type,
12     FLOAT_type,
13     FUNC,
14     MAIN,
15     PROTO,
16 } tokenType;
17 typedef enum
18 {
19     Return_VOID,
20     Return_INT,
21     Return_FLOAT
22 } tokenReturnType;
23
24 typedef enum
25 {
26     CONST_type,
27     VAR_type,
28     NONE_type
29 } tokenConstType;
30
31 struct tokenList
32 {
33     char *name;
34     tokenType type;
35     tokenReturnType returnType;
36     long size;
37     char *scope;
38     long line;
39     long parameter;
40
41     struct tokenList *next;
42 };
43
44 typedef struct tokenList tokenList;

```

```

46
47 struct threeAddressCode
48 {
49     char *code;
50     int gotoLine;
51
52     struct threeAddressCode *next;
53 };
54
55 typedef struct threeAddressCode threeAddressCode;
56
57 struct backPatchList
58 {
59     threeAddressCode *entry;
60     struct backPatchList *next;
61 };
62
63 typedef struct backPatchList backPatchList ;
64
65 threeAddressCode* appendCode(char *code);
66 void backpatch(backPatchList * list, int gotoL);
67
68 backPatchList* mergelists(backPatchList * a, backPatchList * b);
69
70 backPatchList* appendToBackPatch(backPatchList * list, threeAddressCode * entry);
71
72 tokenList* appendToSymbolTable(char *name, tokenType type, tokenReturnType returnType, long size, long line, char *scope, long parameter);
73 void writeCode(FILE *lcoOut);
74 void writeSymbolTable(FILE *symOut);
75
76 tokenList * addSymbolToParameterQueue(tokenList * queue, char *name, tokenType type);
77
78 int checkAndGenerateParams(tokenList * queue, char* name ,int parameterCount);
79
80 int getFunctionType(char *name);
81 int getSymbolType(char *name);
82 char* nextFloatVar();
83 char* nextIntVar();
84 char* nextBoolVar();
85 int nextquad();
86
87 void addFunction(char *name, unsigned int parameter_count, tokenReturnType ret_type, int line);
88 tokenList * lookup(char *name);
89 extern tokenList *symbolTable;
90

```

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

```
1#!/bin/sh
2lex lexicalAnalyzer.l
3yacc -d syntaxChecker.y
4gcc lex.yy.c y.tab.c icg.c -w -g
5./a.out input.c
6rm y.tab.c y.tab.h lex.yy.c
```

Sample Input

```
1#include <stdlib.h>
2int main(){
3    int a;
4    a=40;
5    if(a<50){
6        a=60;
7    }
8    else{
9        a=100;
10   }
11   return 1;
12}
13|
```

Symbol Table


```

1//Testcase generate simple expresion statements
2#include<stdio.h>
3int main(){
4    int a;
5    float t;
6    a=10;
7    t=t*10;
8    return 1;|
9}

```

```

1
2                                     Intermediate Code Generated
3                                     -----
40      GOTO 1
51      a := 10
62      Tf_1 := t * 10
73      t := Tf_1
84      RETURN 1

```

Case2.c

```

1//testcase to generate control statments|
2#include<stdio.h>
3int main(){
4    int a;
5    float t;
6    a=20;
7    if(a<30){
8        t=3.0;
9    }
10   else{
11       t=4.0;
12   }
13   return 2;
14}

```

```

1
2                                     Intermediate Code Generated
3                                     -----
40      GOTO 1
51      a := 20
62      IF (a < 30) GOTO 4
73      GOTO 6
84      t := 3.0
95      GOTO 7
106     t := 4.0
117     RETURN 2

```

Case3.c

```

1//testcase to check looping statements
2#include<stdio.h>
3int main(){
4    int i,s;
5    s=0;
6    for(i=0;i<10;i=i+2){
7        s=s+i;
8        while(s>30)
9        {
10           s=10;
11           if(s<10)
12           {
13
14               }
15           else
16           {
17               }
18       }
19   }
20   return 2;
21}

```

```

1
2                                     Intermediate Code Generated
3                                     -----
40      GOTO 1
51      s := 0
62      i := 0
73      IF (i < 10) GOTO 8
84      GOTO 18
95      Ti_1 := i + 2
106     i := Ti_1
117     GOTO 3
128     Ti_2 := s + i
139     s := Ti_2
1410    IF (s > 30) GOTO 12
1511    GOTO 5
1612    s := 10
1713    IF (s < 10) GOTO 15
1814    GOTO 16
1915    GOTO 10
2016    GOTO 10
2117    GOTO 5
2218    RETURN 2

```

Case4.c

```
1//testcase to generate different functions code
2#include<stdio.h>
3int func1(){
4    int s;
5    s=56;
6}
7int func(int x,int y){
8    x=x+y;
9    func1();
10}
11
12int main(){
13    int a;
14    a=20;
15    func1();
16    return 1;
17}
```

```
1
2                                     Intermediate Code Generated
3                                     -----
40    GOTO 3
51    s := 56
62    a := 20
73    RETURN 1|
```

Case5.c

```
1//testcase to check main function errors
2#include<stdio.h>
3int main();
4int main(){
5    int y;
6    y=20;
7}
```

```
1
2                                     Intermediate Code Generated
3                                     -----
40    GOTO 1
51    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.