Intermediate Code Generator for the C Language

National Institute of Technology Karnataka,Surathkal

Date: 28th March 2019

Submitted To

Dr. P.Santhi Thilagam

Dept. of Computer Science and Engineering

NITK Surathkal

Group Members:

1. Hardik Rana (16CO138)
2. Shushant Kumar (16CO143)
3. Anmol Horo (16CO206)

Abstract

This report contains the details of the tasks finished as a part of the Phase four of Compiler design Lab. The front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code. Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Contents

1. Introduction

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. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers. The second part of compiler, synthesis, is changed according to the target machine.It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Code Generation:

If we generate machine code directly from source code then for n target machine we will have n optimisers and n code generators but if we will have a machine independent intermediate code, we will have only one optimiser. Intermediate code can be either language specific (e.g., Bytecode for Java) or language. independent (three-address code).The following are commonly used intermediate code representation:

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.

For example:

a = b + c * d;

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

r1 = c * d;

r2 = b + r1;

a = r2

r being used as registers in the target program.

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.

Yacc Script:

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

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

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

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

Definition section

%{

%}

Rules section

%%

C code section

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

C Program:

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

1.Compile the script using Yacc tool

- **$ yacc –d parser.y -v**

2. Compile the flex script using Flex tool

- **$ lex lexl.l**

3. After compiling the lex file, lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are

generated after compiling the yacc script.

4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options –ll and –ly
   - **$ g++ –w -g y.tab.c -ly -ll -o parser**
5. The executable file is generated, which on running parses the C file given as a command line input
   - **$ ./compiler test.c**
6. The script also has an option to take standard input instead of taking input from a file.

2. Design Of Programs

Code:

The entire code for lexical analysis is broken down into 3 files: scanner.l and table.h. The scanner.l and table.h are same as in previous reports.

YACC Code: parser.y file

```
%{
	#include <bits/stdc++.h>
	using namespace std;
	#include "symboltable.h"
	#include "lex.yy.c"

	void log_check(int,int);
	void log_res(icg_temp* & expr_left, icg_temp* param1,
icg_temp* param2, const string& oper);
	void arith_check(int,int);
	void three_ad_code(icg_temp* & expr_left, icg_temp* param1,
icg_temp* param2, const string& oper);
	void assi_check(int,int);
	void goto_fix(vector<int>&, int);
	void print_icg(string);
	int yyerror(char *msg);

	table_t stable[MAX_SCOPE];
	int declaration_status = 0,loop_status = 0,function_status =
0,function_type,cur_dtype,arg_list[10],arg_length =
0,func_composite_diff=0,assign_right_check = 0,follow_inst =
0,inter_var = 0;
```

```
    vector<string> output_icg_file;

%}

%union{  int dtype;entry_type* node;icg_temp* icg_cont;string*
oper;vector<int>* follow_l;int instruct;}
%token SHORT INT LONG LONG_LONG SIGNED UNSIGNED CONST VOID CHAR
FLOAT CHAR_STAR
%token <node> decc hexc charc floatc STRING ID
%token and_log or_log leq geq eq neq mul_asn div_asn mod_asn
add_asn sub_asn incr decr IF FOR WHILE CONTINUE BREAK RETURN

%type <node> id const arr_ind
%type <oper> asn_opr;
%type <dtype> function_call
%type <instruct> find_next
%type <icg_cont> expr_left ind_expr expr expr_stmt expr_unary
expr_arithm expr_assign arr if_stm for_stm while_stm
composite_stmt stmts sole_stmt s_stmt direct_goto

%left ','
%right '='
%left or_log
%left and_log
%left eq neq
%left '<' '>' leq geq
%left '+' '-'
%left '*' '/' '%'
%right '!'
%nonassoc UM
%nonassoc LTE
%nonassoc ELSE

%start starter

%%

for_stm: FOR '(' expr_stmt find_next expr_stmt find_next expr ')'
{loop_status = 1;} direct_goto find_next s_stmt {loop_status =
0;}
```

```
              {
                   $$ = new icg_temp();
                   goto_fix($5->follow_if_correct,$11);
goto_fix($12->follow_l,$6); goto_fix($12->follow_cont, $6);
goto_fix($10->follow_l, $4);
                   $$->follow_l =
append_icg($5->follow_if_wrong,$12->follow_br);
                   print_icg(string("goto ") + to_string($6));
               }
           ;

while_stm: WHILE find_next '(' expr  ')' find_next {loop_status =
1;} s_stmt {loop_status = 0;}
              {
                   $$ = new icg_temp();
                   goto_fix($8->follow_l,$2);
goto_fix($4->follow_if_correct,$6);  goto_fix($8->follow_cont,
$2);
                   $$->follow_l =
append_icg($4->follow_if_wrong,$8->follow_br);
                   print_icg(string("goto ") + to_string($2));
              }
          ;

expr_stmt: expr ';'     {
                             $$ = new icg_temp();
                             $$->follow_if_correct =
$1->follow_if_correct; $$->follow_if_wrong = $1->follow_if_wrong;
                        }
                   | ';'    {    $$ = new icg_temp();    }
              ;

expr: expr ',' ind_expr
                   {
                        $$ = new icg_temp();
                        $$->follow_if_correct =
$3->follow_if_correct; $$->follow_if_wrong = $3->follow_if_wrong;
                   }
          | ind_expr
                   {
```

```
                        $$ = new icg_temp();
                        $$->follow_if_correct =
$1->follow_if_correct; $$->follow_if_wrong = $1->follow_if_wrong;
                   }
              ;

ind_expr: ind_expr '>' ind_expr {
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" > "));}
         | ind_expr '<' ind_expr {
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" < "));}
         | ind_expr eq ind_expr {
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" == "));}
         | ind_expr neq ind_expr{
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" != "));}
         | ind_expr geq ind_expr{
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" >= "));}
         | ind_expr leq ind_expr{
log_check($1->dtype,$3->dtype); $$ = new icg_temp();
log_res($$, $1, $3, string(" <= "));}
         | ind_expr and_log find_next ind_expr {
                   log_check($1->dtype,$4->dtype);
                   $$ = new icg_temp();
                   $$->dtype = $1->dtype;
                   goto_fix($1->follow_if_correct,$3);
                   $$->follow_if_correct = $4->follow_if_correct;
                   $$->follow_if_wrong =
append_icg($1->follow_if_wrong,$4->follow_if_wrong);
              }
         | ind_expr or_log find_next ind_expr{
                   log_check($1->dtype,$4->dtype);
                   $$ = new icg_temp();
                   $$->dtype = $1->dtype;
                   goto_fix($1->follow_if_wrong,$3);
                   $$->follow_if_correct =
append_icg($1->follow_if_correct,$4->follow_if_correct);
```

```
                  $$->follow_if_wrong = $4->follow_if_wrong;
              }
         | '!' ind_expr {   $$ = new icg_temp();   $$->dtype =
$2->dtype;    $$->follow_if_correct = $2->follow_if_wrong;
$$->follow_if_wrong = $2->follow_if_correct;}
         | expr_arithm {    $$ = new icg_temp();   $$->dtype =
$1->dtype;    $$->addr = $1->addr;}
     | expr_assign {    $$ = new icg_temp();   $$->dtype =
$1->dtype;}
         | expr_unary   {   $$ = new icg_temp();   $$->dtype =
$1->dtype;}
    ;

expr_unary:   id incr   {    $$ = new icg_temp();   $$->dtype =
$1->dtype;    $$->code = string($1->lexeme) + string("++");
print_icg($$->code);}
           | id decr   {    $$ = new icg_temp();   $$->dtype =
$1->dtype;    $$->code = string($1->lexeme) + string("--");
print_icg($$->code);}
          | decr id    {    $$ = new icg_temp();   $$->dtype =
$2->dtype;    $$->code = string("--") + string($2->lexeme);
print_icg($$->code);}
          | incr id {  $$ = new icg_temp();   $$->dtype =
$2->dtype;    $$->code = string("++") + string($2->lexeme);
print_icg($$->code);}


expr_assign : expr_left asn_opr expr_arithm   {
assi_check($1->node->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $3->dtype; $$->code = $1->node->lexeme + *$2 +
$3->addr;
                  print_icg($$->code);   assign_right_check = 0;}
         | expr_left asn_opr arr{
assi_check($1->node->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $3->dtype; $$->code = $1->node->lexeme + *$2 +
$3->code;print_icg($$->code);   assign_right_check = 0;}
         | expr_left asn_opr function_call {
assi_check($1->node->dtype,$3); $$ = new icg_temp();   $$->dtype
= $3;}
              | expr_left asn_opr expr_unary  {
```

```
assi_check($1->node->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $3->dtype; $$->code = $1->node->lexeme + *$2 +
$3->code;print_icg($$->code);   assign_right_check = 0;}
              | expr_unary asn_opr expr_unary {
assi_check($1->dtype,$3->dtype);$$ = new icg_temp();   $$->dtype
= $3->dtype;  $$->code = $1->code + *$2 + $3->code;
print_icg($$->code);   assign_right_check = 0;}
    ;

asn_opr:'='   {assign_right_check=1; $$ = new string(" = ");}
|add_asn {assign_right_check=1; $$ = new string(" += ");}
    |sub_asn  {assign_right_check=1; $$ = new string(" -= ");}
|mul_asn {assign_right_check=1; $$ = new string(" *= ");}
    |div_asn  {assign_right_check=1; $$ = new string(" /= ");}
|mod_asn {assign_right_check=1; $$ = new string(" %= ");}
    ;

expr_arithm: expr_arithm '+' expr_arithm {
arith_check($1->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $1->dtype; three_ad_code($$, $1, $3, string(" + "));}
              | expr_arithm '-' expr_arithm {
arith_check($1->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $1->dtype; three_ad_code($$, $1, $3, string(" - "));}
              | expr_arithm '*' expr_arithm {
arith_check($1->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $1->dtype; three_ad_code($$, $1, $3, string(" * "));}
              | expr_arithm '/' expr_arithm {
arith_check($1->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $1->dtype; three_ad_code($$, $1, $3, string(" / "));}
             | expr_arithm '%' expr_arithm {
arith_check($1->dtype,$3->dtype);    $$ = new icg_temp();
$$->dtype = $1->dtype; three_ad_code($$, $1, $3, string(" % "));}
              |'(' expr_arithm ')' { $$ = new icg_temp();
$$->dtype = $2->dtype; $$->addr = $2->addr;   $$->code =
$2->code;}
          |'-' expr_arithm %prec UM { $$ = new icg_temp();
$$->dtype = $2->dtype; $$->addr = "t" + to_string(inter_var);
string expr = $$->addr + " = " + "minus " + $2->addr;  $$->code =
$2->code + expr;  inter_var++;}
         |id { $$ = new icg_temp();   $$->dtype = $1->dtype;
```

```
$$->addr = $1->lexeme;}
          |const {  $$ = new icg_temp();   $$->dtype = $1->dtype;
$$->addr = to_string($1->value);}
           ;

const: decc {$1->is_constant=1; $$ = $1;} | hexc
{$1->is_constant=1; $$ = $1;}| charc {$1->is_constant=1; $$ =
$1;}| floatc  {$1->is_constant=1; $$ = $1;};

type : dtype pointer    {declaration_status = 1; }| dtype
{declaration_status = 1; };

pointer: '*' pointer| '*';

dtype : sign_specifier type_specifier | type_specifier;

sign_specifier : SIGNED | UNSIGNED;

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


argument_list : arguments | ;

arguments : arguments ',' arg | arg ;

arg : type id {
                                  arg_list[arg_length++] =
$2->dtype;
                                  print_icg(string("arg ") +
$2->lexeme);
                             }
    ;

s_stmt:composite_stmt       {$$ = new icg_temp(); $$=$1;}
    |sole_stmt          {$$ = new icg_temp(); $$=$1;}
    ;
```

```
composite_stmt :'{'     {
                        if(!func_composite_diff)current_scope =
create_new_scope();
                        else func_composite_diff = 0;
                   }
                   stmts
                   '}'
                   {    current_scope = exit_scope();    $$ = new
icg_temp();   $$ = $3;}
    ;

stmts:stmts find_next s_stmt    {    goto_fix($1->follow_l,$2);
$$ = new icg_temp();   $$->follow_l = $3->follow_l;
$$->follow_br = append_icg($1->follow_br,$3->follow_br);
$$->follow_cont = append_icg($1->follow_cont,$3->follow_cont);}
    |                                {    $$ = new icg_temp();
}
    ;

sole_stmt :  if_stm     {    $$ = new icg_temp();    $$ = $1;
goto_fix($$->follow_l, follow_inst);}
             |for_stm   {    $$ = new icg_temp();    $$ = $1;
goto_fix($$->follow_l, follow_inst);}
         |while_stm {  $$ = new icg_temp();    $$ = $1;
goto_fix($$->follow_l, follow_inst);}
         |declaration       {$$ = new icg_temp();}
         |function_call ';' {$$ = new icg_temp();}
              |RETURN ';'     {  if(function_status)    {
if(function_type != VOID)   yyerror("return type (VOID) does not
match function type");}
                                  else yyerror("return statement
not inside function definition");
                                  }

              |CONTINUE ';' {    if(!loop_status)
yyerror("Illegal use of continue");
                                       $$ = new icg_temp();
$$->follow_cont = {follow_inst};print_icg("goto _");
                                  }
```

```
               |BREAK ';'      {  if(!loop_status)
{yyerror("Illegal use of break");}
                                      $$ = new icg_temp();
$$->follow_br = {follow_inst};  print_icg("goto _");
                                }

               |RETURN ind_expr ';' { if(function_status){
if(function_type != $2->dtype)  yyerror("return type does not
match function type");}
                                         else yyerror("return
statement not in function definition");}
         ;

if_stm:IF '(' expr ')' find_next s_stmt  %prec LTE {
                   goto_fix($3->follow_if_correct,$5);  $$ = new
icg_temp();  $$->follow_l =
append_icg($3->follow_if_wrong,$6->follow_l); $$->follow_br =
$6->follow_br;    $$->follow_cont = $6->follow_cont;}

          |IF '(' expr ')' find_next s_stmt  ELSE direct_goto
find_next s_stmt {
                   goto_fix($3->follow_if_correct,$5);
goto_fix($3->follow_if_wrong,$9);    $$ = new icg_temp();
vector<int> temp = append_icg($6->follow_l,$8->follow_l);
                   $$->follow_l = append_icg(temp,$10->follow_l);
$$->follow_br = append_icg($10->follow_br,$6->follow_br);
$$->follow_cont = append_icg($10->follow_cont,$6->follow_cont);
              }
     ;

declaration: type  declaration_list ';'
{declaration_status = 0;}| declaration_list ';'| expr_unary ';'

declaration_list: declaration_list ',' sub_decl    |sub_decl ;

sub_decl: expr_assign  |id  |arr ;

expr_left: id      {$$ = new icg_temp(); $$->node = $1;}
   | arr {$$ = new icg_temp(); $$->code = $1->code;}
      ;
```

```
id:ID   {
                    if(declaration_status &&
!assign_right_check){
                      $1 =
insert(stable[current_scope].symbol_table,yytext,INT_MAX,cur_dtyp
e);
                      if($1 == NULL) yyerror("Variable Re
Declared");
                    }
                    else{
                      $1 = search_recursive(yytext);
                      if($1 == NULL) yyerror("Variable not
declared");
                    }

                       $$ = $1;
                }
          ;

arr: id '[' arr_ind ']'{
                            if(declaration_status) {
                                if($3->value <= 0) yyerror("size
of array is not positive");
                                else if($3->is_constant)
$1->array_dimension = $3->value;
                            }
                       else if($3->is_constant){
                            if($3->value > $1->array_dimension)
     yyerror("Array index out of bound");
                            if($3->value < 0)  yyerror("Array
index cannot be negative");
                            }
                       $$ = new icg_temp();   $$->dtype =
$1->dtype;
                       if($3->is_constant)         $$->code =
string($1->lexeme) + string("[") + to_string($3->value) +
string("]");
                       else                        $$->code =
string($1->lexeme) + string("[") + string($3->lexeme) +
```

```
string("]");
                        $$->node = $1;
                   }

arr_ind: const          {$$ = $1;}
            | id        {$$ = $1;};

function_call: id '(' parameter_list ')'{ $$ = $1->dtype;
check_parameter_list($1,arg_list,arg_length); arg_length = 0;
print_icg(string("function call ") + $1->lexeme);}
             | id '(' ')'   {    $$ = $1->dtype;
check_parameter_list($1,arg_list,arg_length); arg_length = 0;
print_icg(string("function call ") + $1->lexeme);}
         ;

parameter_list:parameter_list ','  parameter
              |parameter
              ;

parameter: ind_expr     {    arg_list[arg_length++] = $1->dtype;
print_icg(string("param ") + $1->addr);}
          | STRING{    arg_list[arg_length++] = STRING;
print_icg(string("param ") + $1->lexeme);}
          ;

find_next:              {$$ = follow_inst;}
 ;

direct_goto:{ $$ = new icg_temp; $$->follow_l = {follow_inst};
print_icg("goto _");}
     ;


function: type id {function_type = cur_dtype; declaration_status
= 0; current_scope = create_new_scope();  print_icg($2->lexeme +
string(":"));}
          '(' argument_list ')'
              {    declaration_status = 0;
fill_parameter_list($2,arg_list,arg_length);  arg_length = 0;
function_status = 1;   func_composite_diff=1;}
```

```
          composite_stmt    {    function_status = 0;     }
         ;

starter: starter builder
              | builder;

builder: function
              | declaration
              ;


%%

void log_res(icg_temp* & expr_left, icg_temp* param1, icg_temp*
param2, const string& oper)
{
     expr_left->dtype = param1->dtype;
     expr_left->follow_if_correct = {follow_inst};
     expr_left->follow_if_wrong = {follow_inst + 1};
     string code;
     code = string("if ") + param1->addr + oper + param2->addr +
string(" goto _");print_icg(code);
     code = string("goto _");    print_icg(code);
}

void three_ad_code(icg_temp* & expr_left, icg_temp* param1,
icg_temp* param2, const string& oper)
{
     expr_left->addr = "t" + to_string(inter_var);
     string expr = expr_left->addr + string(" = ") + param1->addr
+ oper + param2->addr;
     expr_left->code = param1->code + param2->code + expr;
     inter_var++; print_icg(expr);
}

void goto_fix(vector<int>& vec, int num){
     int n = vec.size();
     for(int i = 0; i<n; i++){
          string instruction = output_icg_file[vec[i]];
          if(instruction.find("_") < instruction.size()) {
```

```
instruction.replace(instruction.find("_"),1,to_string(num));
output_icg_file[vec[i]] = instruction;}
     }
}

void print_icg(string x)
{
     string instruction;
     instruction = to_string(follow_inst) + string(":> ") + x;
     output_icg_file.push_back(instruction);   follow_inst++;
}

void log_check(int left, int right)
{
     if(left != right) yyerror("Mismatch of data types in logical
expr");
}

void arith_check(int left, int right)
{
     if(left != right) yyerror("Mismatch of data types in
arithmetic expr");
}
void assi_check(int left, int right)
{
     if(left != right) yyerror("Mismatch of data types in
assignment expr");
}

void displayICG()
{
     ofstream outfile("INT_CODE_GEN.code");
     for(int i=0; i<output_icg_file.size();i++)
     outfile << output_icg_file[i] <<endl;
     outfile << follow_inst << ":> exit";
     outfile.close();

}

int main()
```

```
{
      int i;
      for(i=0; i<MAX_SCOPE;i++)
      {
       stable[i].symbol_table = NULL;
       stable[i].parent = -1;
      }

     constant_table = create_table();
  stable[0].symbol_table = create_table();
     yyin = fopen("test.c", "r");

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

     displayICG();

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

}

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

Explanation

There are three functions to check the type of the arithmetic expressions, assignment expressions and logical expressions by checking the left hand and right hand side operand.

Three Address Code generation was implemented using the functions **three_ad_code()**. This function takes the the two operands with the intermediate operator keeping operator precedence in consideration. Then `expr_left->addr = "t" + to_string(inter_var)` takes the next temporary variable and uses this variable to store the intermediate value of the arithmetic expression. Then assigning of the value to this temporary variable is done using `string expr = expr_left->addr + string(" = ") + param1->addr + oper + param2->addr` statement. Then the ICG code is updated and the value of the temporary variable number is increased for the next iteration in `inter_var++; print_icg(expr)`.

Backpatching was implemented using the function **goto_fix()**. This function takes the current ICG vector and then travels in it to find the goto _ so that _ can be replaced by the exact number. This concept was implemented as `string instruction = output_icg_file[vec[i]]; if(instruction.find("_")<instruction.size())` then was replaced by finding "_" in it by `instruction.replace(instruction.find("_"),1,to_string(num))` and then finally it is restored to its original ICG code with the correct number as passed above in the production rules `output_icg_file[vec[i]] = instruction`. By using backpatching all the previously unknown goto statements are given the correct flow of instruction direction so that the correct flow in case of IF ELSE, FOR loops and WHILE loop the correct next instruction in both true and false case can be known and be replaced accordingly.

In conditional statements **log_res()** was used to place the correct goto statements for true case and false case. First data type is obtained using `expr_left->dtype = param1->dtype` and then the next two instructions are used to mention the true and false goto statements direction respectively. `expr_left->follow_if_correct={follow_inst},expr_left->follow_if_wrong = {follow_inst + 1}`. Then the conditional statement is printed in the ICG code by adding two goto statements one for guiding to true case and other one to the false case. `code = string("if ") + param1->addr + oper + param2->addr + string(" goto _"); print_icg(code); code = string("goto _"); print_icg(code);`

So in this way the functionalities of Intermediate Code Generation phase were implemented and tested for the intermediate codes accordingly.

Features

- Backward compatibility with Semantic Analyser from Project-3
- Code generation for arithmetic expressions
- Goto fixing for if-else statements and nested if-else
- Goto fixing for while and for loops
- Array indexing
- Goto fixing for logical and relational expressions
- Jumps for break and continue

3. Test Cases

Without Errors:

S.No	Test Case	Expected Output	Status

1	void change()
	{
	int a = 2 + 3+ 4;
	}
	
	void main()
	{
	int j=9;
	change();
	}
		0:> change:
		1:> t0 = 2 + 3
		2:> t1 = t0 + 2
		3:> a = t1
		4:> main:
		5:> j = 9
		6:> call change
		7:> exit
			PASS

2	#include <stdio.h>
	void main()
	{
	int a= 5;
	int b= 6 ;
	if(a<=7)
	b = b - 4;
	else
	b = b + 3;
	}
		0:> main:
		1:> a = 5
		2:> b = 6
		3:> if a <= 7 goto 5
		4:> goto 8
		5:> t0 = b - 4
		6:> b = t0
		7:> goto 10
		8:> t1 = b + 3
		9:> b = t1
		10:> exit
			PASS

p	#include <stdio.h> void main() {	0:> main: 1:> a = 10 2:> b = 2	PASS

	int a = 10;
	int b = 2;
	int c = a + b * 3 / 5;
	}
		3:> t0 = b * 3
		4:> t1 = t0 / 5
		5:> t2 = a + t1
		6:> c = t2
		7:> exit	

Test Case 1:

```
void change()
{
        int a = 2 + 3+ 4;
}
void main()
{
        int j=9;
        change();
}
```

Output 1:

```
0:> change:
1:> t0 = 2 + 3
2:> t1 = t0 + 2
3:> a = t1
4:> main:
5:> j = 9
6:> call change
7:> exit
```

Test Case 2:

```
#include <stdio.h>
void main(){
        int a = 10;
        int b = 2;
        int c = a + b * 3 / 5;
}
```

Output 2 :

```
0:> main:
1:> a = 10
2:> b = 2
3:> t0 = b * 3
4:> t1 = t0 / 5
5:> t2 = a + t1
6:> c = t2
7:> exit
```

Test Case 3:

```
#include <stdio.h>
void main()
{
        int a= 5;
        int b= 6 ;
        if(a<=7)
                b = b - 4;
        else
                b = b + 3;
}
```

Output 3 :

```
0:> main:
1:> a = 5
2:> b = 6
3:> if a <= 7 goto 5
4:> goto 8
5:> t0 = b - 4
6:> b = t0
7:> goto 10
8:> t1 = b + 3
9:> b = t1
10:> exit
```

Test Case 4:

```
#include <stdio.h>
int main()
{
        int a=5;
        int b=6;
        while(a<20)
        {
                b=b+1;
                a=a+1;
        }
        return 0;
}
```

Output 4 :

```
0:> main:
```

```
1:> a = 5
2:> b = 6
3:> if a < 20 goto 5
4:> goto 10
5:> t0 = b + 1
6:> b = t0
7:> t1 = a + 1
8:> a = t1
9:> goto 3
10:> exit
```

Test Case 5:

```
#include <stdio.h>
int main()
{
        int a=5;
        int b=6;
        int c=8;
        for(a=9;a!=6;a=a-1)
        {
                b=b+4;
                c=c-1;
        }
        b=b/9;
        return 0;
}
```

Output 5 :

```
0:> main:
1:> a = 5
```

```
2:> b = 6
3:> c = 8
4:> a = 9
5:> if a != 6 goto 10
6:> goto 15
7:> t0 = a - 1
8:> a = t0
9:> goto 5
10:> t1 = b + 4
11:> b = t1
12:> t2 = c - 1
13:> c = t2
14:> goto 7
15:> t3 = b / 9
16:> b = t3
17:> exit
```

Test Case 6:

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

```
}
```

Output 6 :

```
0:> main:
1:> a = 5
2:> b = 6
3:> if a > 7 goto 5
4:> goto 8
5:> t0 = b + 1
6:> b = t0
7:> goto 3
8:> exit
```

Test Case 7:

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

```
            b=2;
        }
        return 0;
}
```

Output 7 :

```
0:> main:
1:> a = 5
2:> b = 6
3:> if a <= 7 goto 5
4:> goto 13
5:> if a == 9 goto 7
6:> goto 11
7:> t0 = b * 8
8:> b = t0
9:> b = 9
10:> goto 12
11:> a = 10
12:> goto 14
13:> b = 2
14:> exit
```

4. Implementation

The Intermediate Code Generator built for the subset of C language has the following functionalities:

- Backward compatibility with Semantic Analyser from Project-3
- Code generation for arithmetic expressions
- Goto fixing for if-else statements and nested if-else
- Goto fixing for while and for loops
- Array indexing
- Goto fixing for logical and relational expressions
- Jumps for break and continue

5.Results

Results:

The lexical analyzer, syntax analyzer and the semantic analyzer for a subset of C language, which include selection statements, compound statements, iteration statements (for, while and do-while) and user defined functions is generated. It is important to define unambiguous grammar in the syntax analysis phase.

The intermediate Code generator phase generates the three address code for particular grammar.

The lex file (parser.l) and yacc (parser.y) are compiled using following commands:

```
#!/bin/bash

lex lexer.l

yacc -d parser.y -v

g++ -w -g y.tab.c -ly -ll -o semantic_analyser

./semantic_analyser
```

After parsing, if there are errors then the line numbers of those errors are displayed along with a 'parsing failed' on the terminal. Otherwise, a 'parsing complete' message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors.

6.References

1. Lex and Yacc By John R. Levine, Tony Mason, Doug Brown

2. Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

3. https://en.wikipedia.org/wiki/Symbol_table

4.https://www.cse.iitb.ac.in/~br/courses/cs699-autumn2013/refs/lextut-victor-eijkhout.pdf

5.https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/tutorial-large.pdf

6.http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf - Lex and Yacc

Tutorial by Tom Nieman

7. https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/

8.Jenkins Hash Function on Wikipedia :

https://en.wikipedia.org/wiki/Jenkins_hash_function#one-at-a-time