```
//Sam Dressler
//Header for intermediate code generations using 3 address code
#include <stdio.h>
#include <vector>
#include <iostream>
#include <stdlib.h>
#include <unistd.h>
#include <bits/stdc++.h>
#include <tuple>
#include <queue>
using namespace std;

/*
   GLOBAL VARIABLES
*/
//declare the global head for the linked list of 3 address code
struct three_ac_node * three_ac_list_head = NULL;
struct three_ac_node * three_ac_list_tail = NULL;
struct param_list_node * param_list_head = NULL;
struct param_list_node * param_list_tail = NULL;
typedef tuple<string, string, string, string, string> sym_table_entry_t;

sym_table_entry_t bad_entry = make_tuple("bad"," ", " ", " ", " ");
vector<sym_table_entry_t> sym_table;
string current_scope;
string current_scope_id;
stack<string> scope_stack;
bool HELP_ME = true;
int INDEX_VALUE = 0;
```

```
int current_temp_var_num = 0;
int current_label_num = 0;
struct icg_sym_table{
   //Lexeme
   string token_type;
         //Spelling
   string value;
};
typedef struct icg_sym_table icg_symbol;
/*
   Nodes for a linked list where the data section is an identifier and a param count as well as a link to the next param
   param_list_node(id: string, param_num : int, param_list_node * next) -> NULL
     |
     param_node(label: string, param_node * next)
     |
     NULL
*/
struct param_node {
   int param_num = 0;
   string param_type;
   string param_id;
   struct param_node * next;
};
typedef struct param_node param_node;

int param_list_node_count = 0;
struct param_list_node{
   string label;
```

```
   struct param_list_node * next;
   struct param_node * list_head;
   struct param_node * list_tail;
};
typedef struct param_list_node param_list_node;

/*
   Structure that will hold all the three address code generated for statements in program
*/
struct three_ac_node{
   string id;
   int slots_used = 0;
   string three_ac [7];
   // if params string is not "null" then there will be a linked list of parameters containing an identifier and the params
   // the value will be of params in a non "null" situation will be a string for a temporary identifier that will be added to the param list
   param_list_node * params;
   three_ac_node *next;
   three_ac_node *prev;
};
typedef struct three_ac_node three_ac_node;

/*
   Procedure: parses the symbol table and generates three address code for valid statements and indicates an error if one occurs
   @param vector of icg symbols
   @return: void
*/
void generate_three_address_code(vector<icg_symbol>);
```

```
int handle_assignment(FILE * fout, vector<icg_symbol>::iterator dest, vector<icg_symbol>::iterator src,
vector<icg_symbol> icg_sym_table);

/*
   Function generates the 3 address code for cases where the sym type is
   program, procedure, or function,
   @param flag which determines what the sym is
   @param fout_icg where 3 ac is written
   @param iterator with pointer to current symbol type
   @return -1 if error, else returns offset to be added to iterator
*/

int gen_three_ac_prog_proc_func(int flag, FILE * fout_icg, vector<icg_symbol>::iterator it,
vector<icg_symbol>);

/*
   Function used to create a new param list node entry for a function or procedure
*/

string get_param_list_label();

param_list_node * find_param_list(string label);

void add_param_to_list(param_list_node * params, string id, string type, int param_num);

void edit_param_type(param_list_node *params, int param_num, string type);

void write_three_ac_to_fout_icg(FILE* fout, string three_ac[7], int num_filled, int flag);

void add_3_ac_node(string args[7], int num_filled);

void print_list_of_param_lists();

int handle_var_declaration(FILE *, vector<icg_symbol>::iterator , vector<icg_symbol>);

void print_sym_table();

int handle_io(FILE *, vector<icg_symbol>::iterator, vector<icg_symbol>);

int lookup(string, int);

int set_var_value(string, string);

string get_temp_var();
```

```
string type_of_var(string);
int handle_if_then(FILE *, vector<icg_symbol>::iterator, vector<icg_symbol>);
sym_table_entry_t get_sym_table_entry(string );
bool evaluate_condition(vector<icg_symbol>);
int handle_while(FILE *, vector<icg_symbol>::iterator, vector<icg_symbol>);
string get_var_value(string id);
int store_array(FILE * fout_icg,string name, string type, int bound1, int bound2);
string trim_string(string string);
string handle_array_indexing(string id);
int get_array_index_identifier_value(string id);
//////////////////////////////
void print_sym_table(){
   FILE * fout = fopen("output_sym_table.txt", "w");
   for(auto& tuple: sym_table){
      printf("ID : %-12s | SCOPE : %-10s | SCOPE ID : %-12s | VALUE : %-20s | TYPE : %-10s \n",
            get<0>(tuple).c_str(),get<1>(tuple).c_str(),
            get<2>(tuple).c_str(),get<3>(tuple).c_str(),
            get<4>(tuple).c_str());
      fprintf(fout, "%-10s | %-10s | %-12s | %-20s | %-10s \n",
            get<0>(tuple).c_str(),get<1>(tuple).c_str(),
            get<2>(tuple).c_str(),get<3>(tuple).c_str(),
            get<4>(tuple).c_str());
   }
   fclose(fout);
   return;
}
/*
   Procedure: prases the symbol table and generates three address code for valid statements
            indicates an error if one occurs
```

```
    @param vector of icg symbols
    @return: void
*/
void generate_three_address_code(vector<icg_symbol> icg_sym_table){
    int flag = -1;
    int offset = 0;
    icg_symbol temp;
    icg_symbol temp_look_ahead;
    icg_symbol temp_look_behind;
    FILE * fout_icg = fopen("output_icg.txt", "w");
    vector<icg_symbol>::iterator it;
    vector<icg_symbol>::iterator it_prev;
    vector<icg_symbol>::iterator it_next;
    for(it = icg_sym_table.begin(); it != icg_sym_table.end(); ++it){
        temp = *it;

        string t_type = temp.token_type;
        string v_value = temp.value;
        if(it != icg_sym_table.begin()){
            it_prev = it-1;
            temp_look_behind = *it_prev;

        }
        if((it+1) != icg_sym_table.end()){
            it_next = it+1;
            temp_look_ahead = *it_next;
        }
            /*
                Handle Program, Procedure or function declarations
```

```
*/
if(t_type.compare("program_sym") == 0){
   current_scope = "program";
   flag = 0;
   offset = gen_three_ac_prog_proc_func(flag, fout_icg, it, icg_sym_table);
   if(offset == -1){exit(-1);}
   it += offset-1;
}
else if(t_type.compare("procedure_sym") == 0){
   current_scope = "procedure";
   flag = 1;
   offset = gen_three_ac_prog_proc_func(flag, fout_icg, it, icg_sym_table);
   if(offset == -1){exit(-1);}
   it += offset-1;
}
else if(t_type.compare("function_sym")== 0 ) {
   current_scope = "fucntion";
   flag = 2;
   offset = gen_three_ac_prog_proc_func(flag, fout_icg, it, icg_sym_table);
   if(offset == -1){exit(-1);}
   it += offset-1;
}
//Handle begin sym for programs, procedures, or functions,
//Note this does not handle begin or end symbols involved in loops
else if(t_type.compare("begin_sym")==0){
   current_scope_id = scope_stack.top();
   cout << "Current Scope : " << current_scope_id << endl;
   string tac[7];
   tac[0] = "begin";
```

```
    tac[1] = current_scope_id;
   // cout << "t0" << tac[0] << "t1" << tac[1] << endl;
    write_three_ac_to_fout_icg(fout_icg,tac,2,8);
}
else if(t_type.compare("end_sym") == 0){
    string temp_s = scope_stack.top();
    cout << "Leaving Scope : " << current_scope_id << endl;
    string tac[7];
    tac[0] = "end";
    tac[1] = current_scope_id;
    scope_stack.pop();
    if(!scope_stack.empty()){
       current_scope_id = scope_stack.top();
    }
    // cout << "t0" << tac[0] << "t1" << tac[1] << endl;
    write_three_ac_to_fout_icg(fout_icg, tac, 2, 9);
    //cout << "Current Scope : " <<current_scope_id << endl;
}
/*
    Handle varaible declarations
*/
else if(t_type.compare("var_sym") == 0){
    cout << "Defining Variables in Scope : " << current_scope_id << endl;
    offset = handle_var_declaration(fout_icg, it, icg_sym_table);
    if(offset == -1){exit(-1);}
    it += offset-1;
}
/*
    Handle Assignment statements
```

```
        -Take in a icg_symbol table entry and produce 3 address code for it
    */
    else if(t_type.compare("assign") == 0){
        //cout << temp_look_behind.value <<" "<< temp.value << " " <<temp_look_ahead.value
<<endl;
        handle_assignment(fout_icg, it_prev, it_next, icg_sym_table);
        // vector<string> expr = vector<string>();
        // expr.push_back()
    }

    /*
        Handle read/wriite I/O calls
    */
    else if((t_type.compare("write_sym") == 0) || (t_type.compare("read_sym") == 0)){
        offset = handle_io(fout_icg, it, icg_sym_table);
        it += offset-1;
    }
    else if((t_type.compare("writeln_sym") == 0) || (t_type.compare("readln_sym") == 0)){
        offset = handle_io(fout_icg, it, icg_sym_table);
        it += offset-1;
    }
    /*
    *   Handle if then statements
    */

    else if(t_type.compare("if_sym") == 0){
        offset = handle_if_then(fout_icg, it, icg_sym_table);
        it += offset-1;
    }
```

```
        /*
        *   Handle While Loops
        */
        else if(t_type.compare("while_sym") == 0){
            offset+= handle_while(fout_icg, it, icg_sym_table);
            it += offset-1;

        }
        else if(t_type.compare("period") == 0){
            cout << "-----------------------------------------------------"<<endl;
            cout << "Intermediate Code Generation Complete" << endl;
        }
        else if(t_type.compare("semicolon")==0){

        }
        else if(t_type.compare("illegal") == 0){
            cout << "ERROR : unrecognized token" << endl;
        }
        else{
            // cout << "ELSE " << temp.token_type << " : " << temp.value << endl;
        }
    }
    fclose(fout_icg);
}
string handle_array_indexing(string id){
    cout << "ID sent to handle array indexing : " <<id << endl;
    int idx_value = -1;
    idx_value = get_array_index_identifier_value(id);
    //put the idx_value back between the brackets and check if that value exisists in the symbol table
```

```
    size_t idx;
    idx = id.find_first_of('[');

    id.erase(id.begin()+idx, id.end());
    id += ("[" + to_string(idx_value) + "]");
    // cout << id << endl;
    lookup(id, 1);
    if(lookup(id,1) != -1){
        // set_var_value(id, get_var_value(id); //set arr[idx] -> arr[1] to arr[1];
        return id;
    }else{
        cout << "ERROR : Index out of bounds or variable does not exist" << endl;
        exit(-1);
    }
}
int get_array_index_identifier_value(string id){
    size_t idx;
    string index_id;
    // cout << "ARRAY ID : " << id<< endl;
    idx = id.find_first_of('['); //searching for [
    idx++;
    while(true){
        if(id.at(idx)== ']'){
            break;
        }
        index_id += id.at(idx);
        idx++;
    }
    // cout << "idx : " << index_id << endl;
```

```
    string return_val = get_var_value(index_id);
    if(return_val.compare("bad") == 0){
        cout << "ERROR : undefined reference to " << id << endl;
        exit(-1);
    }
    int index_value = atoi(return_val.c_str());
    // cout << "INDEX AS INT " << index_value << endl;
    return index_value;
}
int handle_while(FILE * fout_icg, vector<icg_symbol>::iterator it, vector<icg_symbol>icg_symbol_table){
    cout << "--------------------------------------------------\n";
    cout << "          Handling While Loops\n";
    vector<icg_symbol>::iterator loop_start; // Once the condition is evaluated the pointer is set to the
index after the conditional
    vector<icg_symbol>::iterator it_prev;
    vector<icg_symbol>::iterator it_next;
    vector<icg_symbol> cond_expr = vector<icg_symbol>();
    icg_symbol sym;
    string label;
    string sym_value;
    int offset = 0;
    int offset2 = 0;
    sym = *it;
    while(1){ // get the condition that is being evaluated in this while loop
        it++;offset++;
        sym = *it;
        cond_expr.push_back(sym);
        if(sym.token_type.compare("do_sym")==0){
            cond_expr.push_back(sym);
```

```
        it++;offset++;
        sym = *it;
        break;
      }
   }
   cout << "AFTER - CURR SYM " << sym.value << endl;
   for(vector<icg_symbol>::iterator itx = cond_expr.begin(); itx < cond_expr.end();itx++){
      icg_symbol temp = *itx;
      // cout << temp.token_type << " " << temp.value << endl;
   }
   current_label_num++;//increment the number of labels
   label.append("L"+to_string(current_label_num));// print a label to the file that indicates the start of the while loop statements
   fprintf(fout_icg, "%s\n", label.c_str());
   if(sym.token_type.compare("begin_sym")==0){
      it++;offset++;
      sym = *it;
      loop_start = it;
      INDEX_VALUE = 1;
      if((evaluate_condition(cond_expr)) == true){
         cout << "RETURNED TRUE "<< endl;
         while(true){
            // cout <<"IN WHILE : "<< sym.value << " : "<<sym.token_type << endl;
            while(true){
               if(sym.token_type.compare("end_sym") == 0){
                  cout <<"REACHED END: "<< sym.value << " : "<<sym.token_type << endl;
                  it++;offset++;sym = *it;
                  break;
               }
```

```
            else if((sym.token_type.compare("identifier")==0) ||
sym.token_type.compare("litchar")==0){
                cout << "SYM = ID" << endl;
                // print_sym_table();
                // exit(0);
                it++;offset++;
                sym = *it;
                if(sym.token_type.compare("assign")==0){
                   cout << "SYM = ASSIGN" << endl;
                   if(it != icg_symbol_table.begin()){
                      it_prev = it-1;
                   }
                   if((it+1) != icg_symbol_table.end()){
                      it_next = it+1;
                   }
                      cout << "SYM before Assign : " << sym.token_type << endl;
                      offset2 += handle_assignment(fout_icg, it_prev, it_next, icg_symbol_table);
                      offset+= offset2;
                      //set the location to the original location plus the gained offset
                      it += offset2 - 1; sym = *it;
                      cout << "Out of assign"<<endl;
                   }

               }
               it++;offset++;sym = *it;
            }
            INDEX_VALUE ++;
            cout << "INDEX VALUE == " << INDEX_VALUE << endl;
            if(evaluate_condition(cond_expr)==true){
```

```
            it = loop_start;
            sym = *loop_start;
            cout << "RETURNING TO LOOP START : " << sym.value << endl;
          }
          else{
            it++;offset++;
            break;
          }
        }
      }
      else{
        cout << "Condition on first pass is FALSE" << endl;
      }
    }
    else{
      cout << "ERROR : expected 'begin' before symbol : " << sym.value << endl;
    }
}

int handle_if_then(FILE * fout_icg, vector<icg_symbol>::iterator it,
vector<icg_symbol>icg_symbol_table){
    cout << "--------------------------------------------------\n";
    cout << "          Handling If-Then Statements\n";
    int offset = 0; int offset2 = 0;
    vector<icg_symbol>::iterator start_iterator = it;
    vector<icg_symbol>::iterator it2 = it;
    vector<icg_symbol>::iterator it_prev;vector<icg_symbol>::iterator it_next;

    string cond_string;
```

```
icg_symbol sym = *it2;
vector<icg_symbol> cond_expr = vector<icg_symbol>();
/**
 * We need to determine what the conditional statement is,
 *  evaluate the conditional,
 * and then decide what is written to the icg_symbol table
 */
//PART 1: GET THE first CONDITIONAL i.e. if x = y then S1
while(1){
   cond_string += (sym.value+ " ");
   it2++;offset++;
   sym = *it2;
   cond_expr.push_back(sym);
   if(sym.token_type.compare("then_sym")==0){
      it2++;offset++;
      sym = *it2;
      break;
   }
}
fprintf(fout_icg, "%s\n", cond_string.c_str());
fprintf(fout_icg, "jump else\n");
fprintf(fout_icg, "L%d\n",current_label_num);
current_label_num++;



while(1){
   //get the three address code for the first block of statements
   if(sym.token_type.compare("begin_sym")==0){
      it2++;offset++;
```

```
    sym = *it2;
}
else if((sym.token_type.compare("identifier") == 0) || (sym.token_type.compare("litchar") == 0)){
    it2++;offset++;
    sym = *it2;
    if(sym.token_type.compare("assign") == 0){
        if(it2 != icg_symbol_table.begin()){
            it_prev = it2-1;
        }
        if((it2+1) != icg_symbol_table.end()){
            it_next = it2+1;
        }
        // cout << "SYM before Assign : " << sym.token_type << endl;
        offset2 += handle_assignment(fout_icg, it_prev, it_next, icg_symbol_table);
        offset+= offset2;
        //set the location to the original location plus the gained offset
        it2 += offset2 - 1;
        sym = *it2;
        // cout << "SYM after Assign : " << sym.token_type << endl;
        // cout << "SYM : " << sym.token_type << endl;
    }
    else{
        cout << "ERROR : Unexpected token " << sym.token_type << endl;
        exit(-1);
    }
}
else if(sym.token_type.compare("end_sym") == 0){
    // cout << "END OF FIRST SECTION" << endl;
    it2++;
```

```
        offset++;
        sym=*it2;
        // cout << "FIRST SYM AFTER FIRST SECTION" << sym.token_type<<endl;
        break;
    }
    else{
        cout << "ERROR : Invalid Token : "<< sym.token_type << endl;
        it2++;offset++;
        sym = *it2;
        exit(-1);
    }
}

fprintf(fout_icg, "jump end_if\n");
fprintf(fout_icg, "else\n");
fprintf(fout_icg, "L%d\n",current_label_num);
current_label_num++;
// cout << "HERE AFTER BREAKING FIRST LOOP "<<sym.token_type << endl;
offset2 = 0;
if(sym.token_type.compare("else_sym")==0){
    it2++;offset++;
    sym = *it2;
    while(1){
        //get the three address code for the first block of statements
        if(sym.token_type.compare("begin_sym")==0){
            it2++;offset++;
            sym = *it2;
        }
        else if((sym.token_type.compare("identifier") == 0) || (sym.token_type.compare("litchar") == 0)){
```

```
    it2++;offset++;
    sym = *it2;
    if(sym.token_type.compare("assign") == 0){
        if(it2 != icg_symbol_table.begin()){
            it_prev = it2-1;
        }
        if((it2+1) != icg_symbol_table.end()){
            it_next = it2+1;
        }
        // cout << "SYM before Assign : " << sym.token_type << endl;
        offset2 += handle_assignment(fout_icg, it_prev, it_next, icg_symbol_table);

        offset+= offset2;
        //set the location to the original location plus the gained offset
        it2 += offset2-1; sym = *it2;
        //  cout << "SYM after Assign : " << sym.token_type << endl;
    }
    else{
        cout << "ERROR : Unexpected token " << sym.token_type << endl;
        exit(-1);
    }
}
else if(sym.token_type.compare("end_sym") == 0){
    // cout << "FOUND END" << endl;
    it2++;
    offset++;
    sym=*it2;
    if(sym.token_type.compare("semicolon")==0){
        it2++;
```

```
            offset++;
            sym= *it2;
            break;
          }
          else{
            cout << "ERROR : expected semicolon after final end in if-else statement. Found : " <<
sym.value << endl;
            exit(-1);
          }
        }
        else{
          cout << "ERROR 1: Invalid Token : "<< sym.token_type << endl;
          it2++;offset++;
          sym = *it2;
          // exit(-1);
        }

      }
      // fprintf(fout_icg, "end_if\n");
    }
    fprintf(fout_icg, "jump end_if\n");
    fprintf(fout_icg, "end_if\n");
    fprintf(fout_icg, "L%d\n",current_label_num);
    current_label_num++;
    cout << "            Leaving If-Else Section" << endl;
    cout << "-------------------------------------------------\n";
    return offset;
}
bool evaluate_condition(vector<icg_symbol> cond){
```

```
icg_symbol sym;
sym_table_entry_t temp;
sym_table_entry_t var1; sym_table_entry_t var2;
vector<sym_table_entry_t> vars;
string var_type = "empty";
string op = "empty"; string t_type; string value;
bool first_var = false;
for(vector<icg_symbol>::iterator it = cond.begin(); it < cond.end(); ++it){
   // cout << "here1" << endl;
   sym = *it;
   t_type = sym.token_type;
   value = sym.value;
   if((t_type.compare("litchar") == 0) || (t_type.compare("identifier")==0)){
      // cout << "here2" << endl;
      cout << "VALUE " << value << endl;
      temp = get_sym_table_entry(value);
      cout << "TEMP VAL ID " << get<0>(temp) << endl;



      if(get<0>(temp).compare("bad") == 0){
         cout << "ERROR : Variable not declared : " << value << endl;
         exit(-1);//EXIT since if this outcome occurs the code is ambiguous
      }
      else{
         if(!first_var){
            first_var = true;
         }
         vars.push_back(temp);
```

```
        if(var_type.compare("empty") == 0){
            var_type = get<4>(temp);
        }
        else if(get<4>(temp).compare(var_type)!=0){
            cout << "ERROR : Conflicting types :" <<
            " Type 1 : " << get<4>(temp) << " Type 2 : "<< var_type << endl;
            exit(-1);
        }
    }
}
else if(t_type.compare("number")==0){
    // cout << "here 3" << endl;
    if(!first_var){
        //comparing a variable to a number constant
        //creating a temp sym table entry to compare to the variable
        temp = make_tuple("NUM_CONSTANT","null","null",value, "integer");
        vars.push_back(temp);
    }
}
else if(sym.token_type.compare("equals") == 0){
    if(op.compare("empty") == 0 ){
        op = value;
    }
    else{
        cout << "ERROR : Invalid Operator " << endl;
        exit(-1);
    }
}
else if((t_type.compare("greater") == 0) || (t_type.compare("greaterequal") == 0) ||
```

```
        (t_type.compare("less") == 0) || (t_type.compare("lessequal") == 0) ){
    // cout << "here3 " << endl;
    if(op.compare("empty") == 0 ){
        op = value;

    }
    else{
        cout << "ERROR : Invalid Operator " << endl;
        exit(-1);
    }
}
else if(sym.token_type.compare("true_sym") == 0){
    return true;
}
else if(sym.token_type.compare("false_sym") == 0){
    return false;
}
else if(sym.token_type.compare("then_sym") == 0){
    break;
}
else if(sym.token_type.compare("do_sym")==0){
    // cout << "here 4" << endl;
    break;
}
else
{
    cout << "ERROR : Conditional Statement Invalid" << endl;
    exit(-1);
}
```

```
}
//FOR NOW JUST EVALUATE SIMPLE CONDITIONAL
var1 = vars.front();
var2 = vars.back();
if(op.compare("=") == 0){
   if(get<3>(var1) == get<3>(var2)){
      return true;
   }return false;
}
else if(op.compare(">") == 0){
   if(get<3>(var1) > get<3>(var2)){
      return true;
   }return false;
}
else if(op.compare(">=") == 0){
   if(get<3>(var1) >= get<3>(var2)){
      return true;
   }return false;
}
else if(op.compare("<") == 0){
   if(get<3>(var1) < get<3>(var2)){
      return true;
   }return false;
}
else if(op.compare("<=") == 0){
   if(get<3>(var1) <= get<3>(var2)){
      return true;
   }return false;
   // if(HELP_ME == true){
```

```
      //    if(INDEX_VALUE <= get<3>(var2){
      //       return true;
      //    }
      //    return false;
      // }
      // else{
      //    if(get<3>(var1) <= get<3>(var2)){
      //          return true;
      //       }return false;
      //    }
      // }
   }
}

sym_table_entry_t get_sym_table_entry(string id){
   for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
      sym_table_entry_t temp = *it;
      if ((get<0>(temp).compare(id) == 0)){
         cout << "FOUND ID : " << id << " IN SYMBOL TABLE " << endl;
         cout << "RETURNING OBJECT" << endl;
         return temp;
      }
   }
   return bad_entry;
}
int handle_io(FILE * fout_icg, vector<icg_symbol>::iterator it, vector<icg_symbol> icg_sym_table){
   cout << "-------------------------------------------------"<<endl;
   cout << "          Handling IO Calls" << endl;
```

```
   vector<icg_symbol>::iterator it2 = it;
   icg_symbol sym = *it2;
   int offset = 0;
   int num_filled = 0;
   int flag;
   string tac[7];
   //Case for handling write calls;
   if((sym.token_type.compare("writeln_sym") == 0) || (sym.token_type.compare("write_sym") == 0)){
      flag = 3;
      if(sym.token_type.compare("write_sym") == 0){
         flag = 4;
      }
      tac[0] = sym.value;
      num_filled ++;
      it2++;offset++;
      sym = *it2;
      if(sym.token_type.compare("lparen") == 0){
         it2++;offset++;
         sym = *it2;
         if((sym.token_type.compare("quotestring")==0) || (sym.token_type.compare("identifier") == 0)
||
             (sym.token_type.compare("litchar") == 0)){
            tac[1] = sym.value;
            cout << tac[0] << " " << tac[1] << " " << endl;
            it2++;offset++;
            sym=*it2;
            if(sym.token_type.compare("rparen") == 0){
               it2++;offset++;
               sym=*it2;
```

```
            if(sym.token_type.compare("semicolon") == 0){
               it2++;offset++;
               sym = *it2;
               // cout << "writing tac to fout_icg for write|writeln " <<endl;
               write_three_ac_to_fout_icg(fout_icg, tac, num_filled, flag);
            }
            else{
               cout << "ERROR : expected ';' after statement : actual"<< sym.token_type << endl;
            }
         }
         else{
            cout << "ERROR : expected ')' after string literal" << endl;
            it2++;offset++;
            sym=*it2;
         }
      }
      else if((sym.token_type.compare("identifier") == 0) || (sym.token_type.compare("litchar") == 0)){
         it2++;offset++;
      }
   }
}
//case for handling read or readln
else if((sym.token_type.compare("readln_sym") == 0) || (sym.token_type.compare("read_sym") ==
0)){
   flag = 5;
      if(sym.token_type.compare("read_sym") == 0){
         flag = 6;
      }
      tac[1] = sym.value;
```

```
        num_filled ++;
        it2++;offset++;
        sym = *it2;
        if(sym.token_type.compare("lparen") == 0){
           it2++;offset++;
           sym = *it2;
           if((sym.token_type.compare("quotestring")==0) || (sym.token_type.compare("identifier") == 0)
|| (sym.token_type.compare("litchar") == 0)){
              tac[0] = sym.value;
              tac[2] = sym.value;
              num_filled+=2;
              cout << tac[0] << " = " << tac[1] <<" "<< tac[2] << endl;
              it2++;offset++;
              sym=*it2;
              if(sym.token_type.compare("comma") == 0){
                 it2++;offset++;
                 sym=*it2;
              }
              else if(sym.token_type.compare("rparen") != 0){
                 cout << "ERROR : expected ')' after string literal" << endl;
                 it2++;offset++;
                 sym=*it2;
              }
              else{
                 it2++;offset++;
                 sym=*it2;
                 if(sym.token_type.compare("semicolon") != 0){
                    cout << "ERROR : expected ';' after statement" << endl;
                    it2++;offset++;
```

```
                sym=*it2;
            }
            else{
                // cout << "writing tac to fout_icg for read|readln " <<endl;
                write_three_ac_to_fout_icg(fout_icg, tac, num_filled, flag);

            }
          }
        }
        else if((sym.token_type.compare("identifier") == 0) || (sym.token_type.compare("litchar") ==
0)){
          it2++;offset++;
        }
        else{

        }
      }
    }
  else{
    cout << "ERROR: invalid token" << endl;
    it2++; offset++;
  }
  cout << "-------------------------------------------------"<<endl;
  return offset;
}
int handle_var_declaration(FILE * fout_icg, vector<icg_symbol>::iterator it, vector<icg_symbol>
icg_sym_table){
  cout << "-------------------------------------------------"<<endl;
  cout << "      In var delcaration for "<< current_scope_id << endl;
```

```
cout << "--------------------------------------------------"<<endl;


stack<string> stack;
int offset =0;
vector<icg_symbol>::iterator it2 = it;
icg_symbol sym = *it2;
// cout << sym.value << endl;
// it2++;offset++;
// sym = *it2;
while(true){
   if(sym.token_type.compare("begin_sym") == 0){
      cout << "-- reached end of var section for subroutine --" << endl;
      it2--;offset--;
      break;
   }
   else if(sym.token_type.compare("procedure_sym") == 0){
      break;
   }
   else if(sym.token_type.compare("function_sym") == 0){
      break;
   }
   else if(sym.token_type.compare("var_sym") == 0){
      it2++;offset++;
      sym = *it2;
   }
   else if((sym.token_type.compare("litchar") == 0) || (sym.token_type.compare("identifier") == 0)){
      if(sym.value.compare("var") == 0){
         cout << "ERROR: Using keyword for variable declaration" << endl;
         return -1;
```

```
        }
        stack.push(sym.value);
        it2++;offset++;
        sym = *it2;
        if(sym.token_type.compare("colon")==0){
           it2++;offset++;
           sym = *it2;
           if(sym.token_type.compare("integer_sym") == 0){
              while(!stack.empty()){
                 string temp_id = stack.top();
                 cout << "var " << temp_id << " : integer" << endl;
                 stack.pop();
                                                  //ID      scope type      scope ID        value      type
                 sym_table_entry_t value_temp = make_tuple(temp_id, current_scope, current_scope_id,
"null", "integer");
                 sym_table.push_back(value_temp);
                 // print_sym_table();
              }
              string t = sym.value;
              it2++;offset++;
              sym = *it2;
              if(sym.token_type.compare("semicolon")!= 0){
                 cout << "ERROR expected semicolon after : " << t << endl;
              }
              else{
                 it2++;offset++;
                 sym = *it2;
              }
           }
```

```
        else if(sym.token_type.compare("char_sym") == 0){
            while(!stack.empty()){
                string temp_id = stack.top();
                cout << "var " << temp_id << " : char" << endl;
                stack.pop();
                                        //ID      scope type      scope ID        value      type
                sym_table_entry_t value_temp = make_tuple(temp_id, current_scope, current_scope_id,
"null", "char");
                sym_table.push_back(value_temp);
                // print_sym_table();
            }
            string t = sym.value;
            it2++;offset++;
            sym = *it2;
            if(sym.token_type.compare("semicolon")!= 0){
                cout << "ERROR expected semicolon after : " << t << endl;
            }
            else{
                it2++;offset++;
                sym = *it2;
            }
        }
        else if(sym.token_type.compare("array_sym")==0){
            cout << "var : " << stack.top() << " Type : " << sym.value << endl;
            string array_name = stack.top();
            stack.pop();
            it2++;offset++;
            sym = *it2; // increment pointer to get the lbracket;
            if(sym.token_type.compare("lbrack")==0){
```

```
it2++;offset++;
sym = *it2; //increment pointer to get the first array size
        //Note negative array indexes are not supported at this time
if(sym.token_type.compare("number")==0){
   int array_bound1 = atoi(sym.value.c_str());
   it2++;offset++; //Store the first bound of the array
   sym = *it2;
   if(sym.token_type.compare("period")==0){
      it2++;offset++; //skip the period
      sym = *it2;
      if(sym.token_type.compare("period")==0){
         it2++;offset++; // skip the period
         sym = *it2;
         if(sym.token_type.compare("number")==0){
            int array_bound2 = atoi(sym.value.c_str());//Store the second bound of the array
            it2++;offset++;
            sym = *it2;
            if(sym.token_type.compare("rbrack")==0){
               it2++;offset++;
               sym = *it2;
               if(sym.token_type.compare("of_sym")==0){
                  it2++;;offset++;
                  sym = *it2; // increment pointer to get the array type
                  if(sym.token_type.compare("integer_sym")==0){
                     string array_type = sym.value;
                     it2++; offset++;
                     sym = *it2;
                     if(sym.token_type.compare("semicolon")==0){
```

```
                                    store_array(fout_icg, array_name, array_type, array_bound1,
array_bound2);
                                    it2++;offset++;
                                    sym = *it2;
                                }
                                else{
                                    cout << "ERROR : expected semicolon. Found : " << sym.value << "type :"
<< sym.token_type<< endl;
                                    exit(-1);
                                }
                            }
                            else{
                                cout << "ERROR: expected array type 'integer' or 'int' Found : " <<
sym.value <<endl;
                                cout << "NOTE: only arrays of type 'integer' are currently supported"
<<endl;
                                exit(-1);

                            }
                        }
                        else{
                            cout << "ERROR expected 'of' keyword. Found : " << sym.value << endl;
                            exit(-1);
                        }
                    }
                    else{
                        cout << "ERROR expected ']' after array bounds. Found : " << sym.value << endl;
                        exit(-1);
                    }
                }
```

```
                else {
                    cout << "ERROR: Illegal array spcification :" << sym.value << endl;
                    exit(-1);
                }
            }
            else{
                cout << "ERROR: Illegal array spcification :" << sym.value << endl;
                exit(-1);
            }
        }
        else{
            cout << "ERROR: Illegal array spcification :" << sym.value << endl;
            exit(-1);
        }
    }
    else{
        cout << "ERROR: Illegal array spcification :" << sym.value << endl;
        exit(-1);
    }
  }
  else{
    cout << "ERROR expected '[' after array sym. Found : " << sym.value << endl;
    exit(-1);
  }

}
else{
  cout << "ERROR Invalid type : " << sym.value << endl;
  return -1;
```

```
            }
        }
        else if(sym.token_type.compare("comma")==0){
            it2++;offset++;
            sym = *it2;
        }
    }
    else{
        cout << "ERROR : expected identifier, actual : " <<sym.token_type << " " << sym.value << endl;
        it2++;offset++;
        sym = *it2;
    }
  }
  return offset;

}
int store_array(FILE * fout_icg, string name, string type, int bound1, int bound2){
  string temp_name;
  if(bound1 >= bound2){
    cout << "ERROR : bound 1 greater or equal than bound 2" << endl;
    exit(-1);
  }
  else{
    for(int i = bound1; i <= bound2; i++){
      temp_name = name + "[" + to_string(i) + "]";
      cout << temp_name << " : " << type << endl;
                                    //ID      scope type      scope ID        value      type
      sym_table_entry_t value_temp = make_tuple(temp_name, current_scope, current_scope_id,
"null", "integer");
```

```
            sym_table.push_back(value_temp);
        }
    }
    return 0;
}
//Function that checks the symbol table and returns the type of the ID provided if its in the table
string type_of_var(string s){
    // print_sym_table();
    for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
        sym_table_entry_t temp = *it;
        // cout << "ID : " << get<0>(temp) << endl;
        if(get<0>(temp).compare(s) == 0){
            // cout <<"FOUND var " <<  get<0>(temp)<< " of type : " << get<4>(temp)<<endl;
            return get<4>(temp);
        }
    }
    cout << "ID : " << s << " not found in symbol table " << endl;
    return NULL;
}
int handle_assignment(FILE * fout, vector<icg_symbol>::iterator dest, vector<icg_symbol>::iterator src,
vector<icg_symbol> icg_sym_table){
    cout << "-------------------------------------------------"<<endl;
    cout << "              In assignment" << endl;
    cout << "-------------------------------------------------"<<endl;
    int offset = 0;
    int op_count = 0;
    int lparen_count = 0;
    int rparen_count = 0;
    vector<icg_symbol>::iterator it2 = dest;
```

```
string tac[7];
icg_symbol sym = *it2;
string current_id;
string set_value;
string set_value_type= "null";
string temp_var = "";
string prev_temp_var = "";
string src_type;
string dest_type;
string temp_label;
string sym_value;
bool temp_set = false;
bool prev_temp_set = false;
stack <string > s;

// cout << "sym val: " << sym.value << endl;
   current_id = sym.value;
   it2++;
   sym = *it2;
   while(sym.token_type.compare("semicolon")!=0){
      s.push(sym.value);
      // cout << "SYM TYPE " << sym.token_type << endl;
      if(sym.value == "+" || sym.value == "-" || sym.value == "*" || sym.value == "/"){
         op_count++;
      }
      else if(sym.token_type == "lparen"){
         lparen_count ++;
      }
      else if(sym.token_type == "rparen"){
```

```
            rparen_count++;
        }
        else if(sym.token_type == "number"){
            set_value = sym.value;
            src_type = "integer";
        }
        else if(sym.token_type == "quotechar"){
            set_value = sym.value.at(1);
            src_type = "char";
        }
        else if(sym.token_type == "identifier"){
            set_value = sym.value;
            set_value_type = sym.token_type;
            // cout << "HERE" << endl;
            if(sym.value.find('[') == string::npos){
                src_type = type_of_var(sym.value);
            }
            // cout << "HERE" << endl;
        }
        else if(sym.token_type == "litchar"){
            set_value = sym.value;
            set_value_type = sym.token_type;
            src_type = type_of_var(sym.value);
        }
        // else if((sym.token_type != "litchar") && (sym.token_type != "identifier") && (sym.token_type
!= "mod_sym")){
        //           cout << "ERROR : epected semicolon after expression" << endl;
        //       }
        it2++;
```

```
    sym = *it2;
}
if(lparen_count != rparen_count){
    cout << "ERROR : missing parenthesis in statement" << endl;
    exit(-1);
}
// cout <<endl<< "op_count " << op_count << endl;
string vars[op_count];
// cout << "current_temp_var_num : " << current_temp_var_num << endl;
for(int i = 0; i < op_count; i ++){
    vars[i].append(("tmp"+to_string(current_temp_var_num)));
    prev_temp_var = vars[i];
    lookup(vars[i], 2); // add var to symbol table
    current_temp_var_num++;
}
// int curr_index = (current_temp_var_num-offset);
// cout << "curr_index : " << curr_index<<endl;
it2 = dest;
sym = *it2;
if(op_count != 0){
    set_value.clear();
    while(op_count > 0){
        // cout << set_value << endl;
        // cout << "op_count : " << op_count << endl;
        // cout << "SYM TYPE : " << sym.token_type << endl;
        //skip what we will handle later which is assigning the last temp variable to the identifier
        if((sym.value.compare(":=") == 0)){
            it2++;offset++;
            sym = *it2;
```

```
}
else if((sym.token_type.compare("number") == 0) || (sym.token_type.compare("litchar")== 0)
                                        ||(sym.token_type.compare("identifier")== 0)){
   if((sym.value.find("[") != string::npos) &&(sym.value.find("]") != string::npos)){
      cout << "ARRAY ID : "<< sym.value << endl;
      if(lookup(sym.value,1) != -1){ // check if the index identifier is a number and exists
         cout << "DIRECT ARRAY ACCESSING " << endl; // the id is form <array_var_id>[<n-f>]
                                              // where n is the first index and f is the last
         sym_value = sym.value;
         set_value = sym_value;
         set_value += " ";
      }
      else{
         cout << "INDIRECT ARRAY ACCESSING " << endl;
         sym_value = handle_array_indexing(sym.value);
         // print out the value of the modified index: arr[idx] -> arr[1]
         cout <<"Array Var converted to : "<< sym_value << endl;
         cout << get_var_value(sym_value) << endl;
         set_value = sym_value;
         set_value += sym_value;
         set_value += " ";
      }
   }
   else {
      if(lookup(sym.value, 1) < 0 ){
         cout << "ERROR : undefined reference to : " << sym.value << endl;
         exit(-1);
       }
      cout << "NORMAL ID : " << sym.value << endl;
```

```
    set_value = sym.value;
    set_value += " ";
}

// cout << "SV1: " << set_value << endl;
it2++;offset++;
sym = *it2;
if(sym.value == "+" || sym.value == "-" || sym.value == "*" || sym.value == "/"){
    set_value += sym.value; set_value += " ";
    // cout << "SV2: " << set_value << endl;
    op_count--;
    it2++; offset++;
    sym = *it2;
    if((sym.token_type.compare("number") == 0) || (sym.token_type.compare("litchar")== 0)
                                        ||(sym.token_type.compare("identifier")== 0)){
        // if(get_var_value(sym.value)== "empty") Might do something here ..?
        set_value += sym.value;
        set_value += " ";
        // cout << "SV3 : " << set_value << endl;
        it2++;offset++;
        sym = *it2;
        if(sym.token_type.compare("semicolon")==0){
            it2++;offset++;
            sym = *it2;
            temp_label = get_temp_var();
            // cout << "TEMP LABEL : " << temp_label << endl;
            set_var_value(temp_label, set_value);
            tac[0] = temp_label;
            tac[1] = ":=";
```

```
        tac[2] = set_value;
        write_three_ac_to_fout_icg(fout, tac, 3, 7);
        set_value.clear();
        if(temp_var.compare("") == 0){
          // cout << "here 2" << endl;
          temp_var = temp_label;
        }
        else{
          // cout << "here 3" << endl;
          string swap;
          swap = temp_var;
          temp_var = temp_label;
          prev_temp_var = temp_var;
          prev_temp_set = true;
        }
      }
      else{
        temp_label = get_temp_var();
        cout << temp_label << endl;
        set_var_value(temp_label,set_value);
        set_value = temp_label;set_value += " ";
        // cout << "SV : " << set_value << endl;
      }

  }
  else{
    cout << "ERROR : illegal token in expression : " << sym.value;
    exit(-1);
  }
```

```
    }
    else if(sym.token_type.compare("semicolon")==0){
        src_type = sym.token_type;
    }
}
else if(sym.token_type.compare("lparen") == 0){
    // cout << "HERE 3 " << endl;
    // cout << set_value << endl;
    it2++;offset++;
    sym = *it2;
        if((sym.token_type.compare("number") == 0) ||(sym.token_type.compare("litchar")==0)
                || (sym.token_type.compare("identifier")== 0) || (temp_set == true)){

            set_value = trim_string(set_value);
            if(set_value.compare(current_id) == 0){
                set_value = sym.value;
                set_value += " ";
                // cout << "SV H : " << set_value << endl;
            }
            else{
                set_value += sym.value;
                set_value += " ";
                // cout << "SV J: " << set_value << endl;
            }
                if(temp_set){
                    set_value.clear();
                    set_value += temp_var;
                    set_value += " ";
                }
```

```
                    // cout << "SV4: " << set_value << endl;
                    it2++;offset++;
                    sym = *it2;
                    if(sym.value == "+" || sym.value == "-" || sym.value == "*" || sym.value == "/"){
                       set_value += sym.value; set_value += " ";
                       // cout << "SV5: " << set_value << endl;
                       op_count--;
                       it2++; offset++;
                       sym = *it2;
                       if((sym.token_type.compare("number") == 0) ||
(sym.token_type.compare("litchar")==0)
                       || sym.token_type.compare("identifier")== 0){
                          set_value += sym.value; set_value += " ";
                          // cout << "SV6 : " << set_value << endl;
                          it2++;offset++;
                          sym = *it2;
                          if(sym.token_type.compare("rparen")==0){
                             it2++;offset++;
                             sym = *it2;
                             // cout << "OFFSET " << offset << endl;
                                string temp_label = get_temp_var();
                                // cout << "TEMP VAR : " << temp_label << endl;

                                set_var_value(temp_label, set_value);
                                tac[0] = temp_label;
                                tac[1] = ":=";
                                tac[2] = set_value;
                                write_three_ac_to_fout_icg(fout, tac, 3, 7);
```

```
                    set_value.clear();
                    if(temp_label.compare("") == 0){
                       // cout << "here 2" << endl;
                       temp_label = temp_label;
                    }
                    else{
                       // cout << "here 3" << endl;
                       string swap;
                       swap = temp_var;
                       temp_var = temp_label;
                       prev_temp_var = temp_var;
                       prev_temp_set = true;
                    }
                 }
              }
           }
        }
     // }
  }
  else if((sym.value == "+") ||(sym.value == "-") || (sym.value == "*")|| (sym.value == "/")){
     if(prev_temp_set == true){
        set_value += temp_var; set_value+= " ";
     }
     set_value += sym.value; set_value += " ";
     it2++;offset++;
     sym = *it2;
     op_count --;
     // cout << "SV7 : " << set_value << endl;
     if((sym.token_type.compare("number") == 0) || sym.token_type.compare("identifier")== 0){
```

```
set_value += sym.value; set_value += " ";
// cout << "SV8 : " << set_value << endl;
it2++;offset++;
sym = *it2;
if(sym.token_type.compare("semicolon")==0){
   it2++;offset++;
   sym = *it2;
   // cout << "OFFSET 2 : " << offset << endl;
   // cout << "SYM " << sym.value << endl;
   string temp_label = get_temp_var();
   // cout << "TEMP VAR : " << temp_label << endl;

   set_var_value(temp_label, set_value);
   tac[0] = temp_label;
   tac[1] = ":=";
   tac[2] = set_value;
   write_three_ac_to_fout_icg(fout, tac, 3, 7);
   set_value.clear();
   if(temp_label.compare("") == 0){
      // cout << "here 2" << endl;
      temp_label = temp_label;
   }
   else{
      // cout << "here 3" << endl;
      string swap;
      swap = temp_var;
      temp_var = temp_label;
      prev_temp_var = temp_var;
      prev_temp_set = true;
```

```
                }
            }
        }
    }
    else{
        cout << "expected identifier after assign sym. Actual :" << sym.value << endl;
    }
}

// cout << current_id << " temp var val : " << temp_var << endl;
// cout << current_id << " set val : " << set_value << endl;
set_var_value(current_id , temp_var);
tac[0] = current_id;
tac[1] = ":=";
tac[2] = temp_var;
write_three_ac_to_fout_icg(fout, tac, 3, 7);
}
else{
cout << "SET VAR VAL : " << set_value<< endl;
cout << "SET VAR TYPE : " << set_value<< endl;

if((current_id.find("[") != string::npos) &&(current_id.find("]") != string::npos)){
    cout << "ARRAY ID : "<< current_id << endl;
    if(lookup(current_id,1) != -1){ // check if the index identifier is a number and exists
        cout << "DIRECT ARRAY ACCESSING " << endl; // the id is form <array_var_id>[<n-f>]
                                        // where n is the first index and f is the last
        sym_value = current_id;
        // set_value = sym_value;
        // set_value += " ";
```

```
        }
        else{
            cout << "INDIRECT ARRAY ACCESSING " << endl;
            sym_value = handle_array_indexing(current_id);
            // print out the value of the modified index: arr[idx] -> arr[1]
            cout <<"Array Var converted to : "<< sym_value << endl;
            cout << get_var_value(sym_value) << endl;
            current_id = sym_value;
            // set_value = sym_value;
            // set_value += sym_value;
            // set_value += " ";
        }
    }
    cout << "current ID " << current_id << endl;
    cout << "set val : " << set_value<<endl;
    dest_type = type_of_var(current_id);
    if(dest_type.compare(src_type) == 0){

        if((set_value_type.compare("identifier")==0) || set_value_type.compare("litchar")==0){
            set_value = get_var_value(set_value);
            set_var_value(current_id , set_value); // replace the undefined status in the symbol table with the correct value
            tac[0] = current_id;
            tac[1] = ":=";
            tac[2] = set_value;
            write_three_ac_to_fout_icg(fout, tac, 3, 7); // write the assignment to the TAC fout_icg that will be used for generating code
        }else{
            set_var_value(current_id , set_value); // replace the undefined status in the symbol table with the correct value
```

```
            tac[0] = current_id;
            tac[1] = ":=";
            tac[2] = set_value;
            write_three_ac_to_fout_icg(fout, tac, 3, 7); // write the assignment to the TAC fout_icg that will be used for generating code
        }
        // print_sym_table();
        // cout << "setval : " << set_value << endl;

      }
      else{
        cout << "ERROR : Implicit conversion from " << src_type << " to : " << dest_type << endl;
        exit(-1);
      }

    }
  // cout << offset << endl;
  //Need to do varaible declarations and expressions first
  return offset;
}
string get_temp_var(){
  string ref;
  for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
    sym_table_entry_t temp1 = *it;
    string ref;
    if ((get<1>(temp1).compare("temp") == 0) && get<3>(temp1).compare("null") == 0){
      ref = get<0>(temp1);
      // cout << "REF : " <<ref << endl;
      return ref;
```

```
        }
    }
    return 0;
}
int set_var_value(string id, string value_ptr){
    //ID      scope type      scope ID        value      type
    string idd, scope, scope_id, val, type;
    for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
        sym_table_entry_t temp1 = *it;
        sym_table_entry_t temp2;
        if ((get<0>(temp1).compare(id) == 0)){
            tie(id, scope, scope_id, val, type) = temp1;
            val = value_ptr;
            temp2 = make_tuple(id,scope,scope_id,val,type);
            cout << "New Val Of " << id << " = " <<get<3>(temp2) << endl;
            replace(sym_table.begin(), sym_table.end(), temp1, temp2);
            return 0;
        }
    }
    return 0;
}
string get_var_value(string id){
    //ID      scope type      scope ID        value      type
    string idd, scope, scope_id, val, type;
    // cout << "here" << endl;
    for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
        sym_table_entry_t temp1 = *it;
        if ((get<0>(temp1).compare(id) == 0)){
            // cout << "FOUND : "<<get<3>(temp1) << endl;
```

```
            return get<3>(temp1);
        }
    }
    cout << "not found" << endl;
    return "empty";
}

//ID : variable being looked up
//MODE: 1 - check if variable exists in current scope
//      2 - check to see if temp variables exist, if not push back on vector
int lookup(string id, int mode){
    //
    // cout << "SEARCH VAL : " << id << " CURRENT SCOPE ID: " << current_scope_id << endl;
    switch(mode){
        //check if variable exists in current scope
        case 1:
            for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
                sym_table_entry_t temp = *it;
                if ((get<0>(temp).compare(id) == 0) && (get<2>(temp).compare(current_scope_id)== 0)){
                    // cout << "found var "<< id << " in scope : "<<current_scope_id <<endl;
                    return 0;
                }
                else{
                    // cout << "var: " << id << " could not be found, actual : " << get<0>(temp) << endl;
                    // cout << "expected : " << current_scope_id  << " actual : " << get<2>(temp) << endl;
                }
            }
        break;
        //check to see if temp variables exist, if not push back on vector
```

```
        case 2:
            for(vector<sym_table_entry_t>::iterator it = sym_table.begin(); it < sym_table.end(); ++it){
                sym_table_entry_t temp = *it;
                if ((get<0>(temp).compare(id) == 0)){
                    // cout << "found var "<< id << " in scope : "<<current_scope_id <<endl;
                    return 0;
                }
                else{
                    // cout << "var: " << id << " could not be found, actual : " << get<0>(temp) << endl;
                    // cout << "expected : " << current_scope_id  << " actual : " << get<2>(temp) << endl;
                }
            }
                                    //ID  scope type   scope ID        value       type
            sym_table_entry_t entry = make_tuple(id, "temp", current_scope_id, "null", "integer");
            cout << "Adding " << id << " to symbol table" << endl;
            sym_table.push_back(entry);
            return 0;
        break;
    }
    //check if its in the scope;
    return -1;
}
int gen_three_ac_prog_proc_func(int flag, FILE * fout, vector<icg_symbol>::iterator it,
vector<icg_symbol> icg_sym_table){
    string prog_or_proc;
    int offset = 0;
    bool set = false;
    int flag2 = -1;
    icg_symbol sym = *it;
```

```
//variable string array used for the writing of the string to the output fout_icg
int num_filled = 0;
string three_ac [7];
vector<icg_symbol>::iterator it2 = icg_sym_table.begin();

switch(flag){
   //Program or procedure; has no return value
   //3ac of call p, n (where p is identifier for func, n is args array);
   case 0:
      prog_or_proc = "program";
      set = true;
   case 1:
      if(!set) {prog_or_proc = "procedure";}
      cout << "-------------------------------------------------"<<endl;
      cout <<"Generating 3 address code for call to " << prog_or_proc << "..." << endl;
      cout << "-------------------------------------------------"<<endl;
      // sleep(1);
      while(true){
         icg_symbol temp = *it2;
         //search symbol array for the proc/prog sym
         if(temp.value != sym.value){
            it2++;
         }
         //if its found or its not found and not the end of the input
         else if(it2 != icg_sym_table.end()){
            it2++; offset++;
            sym = *it2;
            // cout << sym.value << endl;
            //Make the first index the call keyword for program and procedures
```

```
three_ac[0] = "call";
//Make the second index the identifier for the procedure or program being called
three_ac[1] = sym.value;
current_scope_id = sym.value;
scope_stack.push(current_scope_id);
num_filled = 2;
it2 ++; offset ++;
sym = *it2;
// cout << sym.value << endl;
//Check if there are any parameters (case handling for procedure);
if(sym.token_type.compare("semicolon") == 0 ){
   three_ac[2] = "null";
   num_filled++;
   flag2 = 0;
}
//else if the next sym after procedure identifier isn't a semicolon
//(means there are parameters);
else{
   stack<string> param_stack;
   int paren_pairs = 0;
   bool lparen_flag = false;
   int current_param_count = 0;
   string param_type;
   string param_id;
   string L = get_param_list_label();
   print_list_of_param_lists();
   // cout << "new Label is : " << L << endl;
   param_list_node * params = find_param_list(L);
   // cout << params->label << endl;
```

```
three_ac[2] = L;
// cout << "three_ac_2" << three_ac[2] << endl;
num_filled++;
//check sym until semicolon is reached and add identifiers or litchars to the param
while(sym.token_type.compare("semicolon") != 0){
   //cout << sym.token_type << endl;
   //cout << "in while" << endl;
   if((sym.token_type.compare("identifier") == 0)
     ||(sym.token_type.compare("litchar") == 0)) {
      //   cout << "sym type = " << sym.token_type << endl;
      param_id = sym.value;
      it2++; offset++;
      sym = *it2;
      //after identifier or lit char , must be a semicolon or comma
      if(sym.token_type.compare("colon") == 0){
         // cout << "sym type = colon" << endl;
         it2++; offset++;
         sym = *it2;
            if(sym.token_type.compare("integer_sym") == 0){
               // cout << "sym type = integer_sym" << endl;
               current_param_count++;
               param_type = "integer";
               add_param_to_list(params, param_id, param_type, current_param_count);
            }
            else if(sym.token_type.compare("real_sym") == 0){
               // cout << "sym type = real_sym" << endl;
               current_param_count++;
               param_type = "real";
               add_param_to_list(params, param_id, param_type, current_param_count);
```

```
            }
            else{
               cout << "error in icg : invalid type declaration" << endl;
            }
      }
      //case for comma
      else if(sym.token_type.compare("comma") == 0){
            param_type = "temp";
            current_param_count++;
            add_param_to_list(params, param_id, param_type, current_param_count);
            it2++; offset++;
            sym = *it2;
            continue;
      }
      else{
         cout << "error: in icg: expected ','";
         cout << "actual value : " << sym.token_type << endl;
      }
}


// else if(sym.token_type.compare("litchar") == 0){
//     cout <<"---" << sym.token_type << " : " << sym.value<< endl;
// }
else if(sym.token_type.compare("lparen") == 0){
   // cout << "sym type = lparen" << endl;
   lparen_flag = true;
}
else if(sym.token_type.compare("rparen") == 0){
   // cout << "sym type = rparen" << endl;
```

```
                if(lparen_flag){
                    paren_pairs++;
                    lparen_flag = false;
                }
                else{
                    cout << "Error : unequal num of parenthesis" << endl;
                }
            }
            // else{
            //     cout << "ERROR : unhandled sym -> " << sym.value<< endl;
            // }
            //Always executed unless "continued"
            it2++; offset++;
            sym = *it2;
        }
        flag2 = 1;
    }
    it2++;offset++;
    break;
  }
  else{
    cout << "ERROR : symbol not found " << endl;
    return -1;
  }
}
//determine if this is needed via testing
// cout << "HERE " << prog_or_proc << endl;
write_three_ac_to_fout_icg(fout, three_ac, num_filled, flag);
return offset;
```

```
   // cout << sym.value << endl;
break;
//Function; has a return value
//3ac of y = call p, n
case 2:
   cout << "--------------------------------------------------"<<endl;
   cout << "Generating 3 address code for call to function..." << endl;
   cout << "--------------------------------------------------"<<endl;
   flag2 = 2;
   // scope_stack.push(current_scope_id);
   // sleep(1);
   // while(true){
   //    icg_symbol temp = *it2;
   //    //search symbol array for the proc/prog sym
   //    if(temp.value != sym.value){
   //       it2++;
   //    }
   //    //if its found or its not found and not the end of the input
   //    else if(it2 != icg_sym_table.end()){
   //       it2++;offset ++;
   //       sym = *it2;
   //       // cout << sym.value << endl;
   //       //0th index is reserved for the return value
   //       three_ac[0] = " "; //space temporary for now.
   //       three_ac[1] = "="; // make the 1st index
   //       three_ac[2] = "call";// add the call keyword
   //       three_ac[3] = sym.value; //add the identifier for the function call
   //       num_filled = 4;
   //       //increment the offset so to find the next parameter;
```

```
//        it2 ++; offset ++;
//        sym = *it2;
//        // cout << sym.value << endl;
//        //Check if there are any parameters (case handling for procedure);
//        if(sym.token_type.compare("semicolon") == 0 ){
//           three_ac[4] = "null";
//           num_filled++;
//           flag2 = 0;
//        }
//        //else if the next sym after procedure identifier isn't a semicolon
//        //(means there are parameters);
//        else{
//           int paren_pairs = 0;
//           bool lparen_flag = false;
//           int current_param_count = 0;
//           string param_type;
//           string param_id;
//           string L = get_param_list_label();
//           param_list_node * params = find_param_list(L);
//           cout << params->label << endl;
//           three_ac[4] = L;
//           num_filled++;
//           //check sym until semicolon is reached and add identifiers or litchars to the param
//           while(sym.token_type.compare("semicolon") != 0){
//              //cout << sym.token_type << endl;
//              //cout << "in while" << endl;
//              if((sym.token_type.compare("identifier") == 0)
//                ||(sym.token_type.compare("litchar") == 0)) {
//                  //   cout << "sym type = " << sym.token_type << endl;
```

```
//                param_id = sym.value;
//                it2++; offset++;
//                sym = *it2;
//                //after identifier or lit char , must be a semicolon or comma
//                if(sym.token_type.compare("colon") == 0){
//                   // cout << "sym type = colon" << endl;
//                   it2++; offset++;
//                   sym = *it2;
//                      if(sym.token_type.compare("integer_sym") == 0){
//                         // cout << "sym type = integer_sym" << endl;
//                         current_param_count++;
//                         param_type = "integer";
//                         add_param_to_list(params, param_id, param_type, current_param_count);
//                      }
//                      else if(sym.token_type.compare("real_sym") == 0){
//                         // cout << "sym type = real_sym" << endl;
//                         current_param_count++;
//                         param_type = "real";
//                         add_param_to_list(params, param_id, param_type, current_param_count);
//                      }
//                      else{
//                         cout << "error in icg : invalid type declaration" << endl;
//                      }
//                }
//                //case for comma
//                else if(sym.token_type.compare("comma") == 0){
//                      param_type = "temp";
//                      current_param_count++;
//                      add_param_to_list(params, param_id, param_type, current_param_count);
```

```
//                  it2++; offset++;
//                  sym = *it2;
//                  continue;
//              }
//              else{
//                  cout << "error: in icg: expected ','";
//                  cout << "actual value : " << sym.token_type << endl;
//                  continue;
//              }
//          }

//          // else if(sym.token_type.compare("litchar") == 0){
//          //     cout <<"---" << sym.token_type << " : " << sym.value<< endl;
//          // }
//          else if(sym.token_type.compare("lparen") == 0){
//              cout << "sym type = lparen" << endl;
//              lparen_flag = true;
//          }
//          else if(sym.token_type.compare("rparen") == 0){
//              cout << "sym type = rparen" << endl;
//              if(lparen_flag){
//                  paren_pairs++;
//                  lparen_flag = false;
//              }
//              else{
//                  cout << "Error : unequal num of parenthesis" << endl;
//              }
//          }
//          // else{
```

```
        //              //    cout << "ERROR : unhandled sym -> " << sym.value<< endl;
        //              // }
        //              //Always executed unless "continued"
        //              it2++; offset++;
        //              sym = *it2;
        //           }
        //           flag2 = 2;
        //       }
        //       it2++;offset++;
        //       break;
        //    }
        //    else{
        //       cout << "ERROR : symbol not found " << endl;
        //       return -1;
        //    }
        // }
        // return offset;
        // cout << sym.value << endl;
        return 1;
      break;
      //case -1:
      default:
        cout << "Error : flag not set" << endl;
        return -1;
      break;
   }
}
void write_three_ac_to_fout_icg(FILE *fout, string tac[7], int num_filled, int mode){
   int i = 0;
```

```
num_filled++;
 param_list_node * temp = new param_list_node;
 param_node *current_param = new param_node;
switch(mode){
   //write 3ac for prog to fout_icg and add to linked list of 3-address codes
   //
   //  TO DO: ADD CALL TO ADD TO LINKED LIST
   //
   case 0:
      tac[num_filled-1] = "program"; // add the id to the next index
      // cout <<"added id : "  << tac[num_filled-1] << endl;
      add_3_ac_node(tac, num_filled);
      for(i = 0; i < num_filled-2; i ++){
         // cout << tac[i] << endl;
         if(i == 1){
            // fprintf(fout, "%s ", tac[i].c_str());
         }
         else{
            // fprintf(fout, "%s ", tac[i].c_str());
         }
      }
      // cout << tac[i] << endl;
         // fprintf(fout, "%s\n",tac[num_filled-2].c_str());
   break;
   //case procedure with parameters
   case 1:
       tac[num_filled-1] = "procedure";
      // cout <<"added id : "  << tac[num_filled-1] << endl;
      temp = find_param_list(tac[2]);//index two is the param list identifier
```

```
    current_param = temp->list_head;
    add_3_ac_node(tac, num_filled);
    if(current_param != NULL){ //if the list node has any params
        // cout << temp->label << " Param : " << current_param->param_id << endl;
        fprintf(fout, "param %s\n", current_param->param_id.c_str());

        while(current_param->next != NULL)
        {
            current_param = current_param->next;
            // cout << temp->label << " Param : " << current_param->param_id << endl;
            fprintf(fout, "param %s\n", current_param->param_id.c_str());

        }
    }
    for(i = 0; i < num_filled-2; i ++){
        // cout << tac[i] << endl;
        if(i == 1){
            fprintf(fout, "%s ", tac[i].c_str());
        }
        else{
            fprintf(fout, "%s ", tac[i].c_str());
        }
    }
    // cout << tac[i] << endl;
    fprintf(fout, "%s\n",tac[num_filled-2].c_str());
break;
//Function case: 3-addresscode has a return label(id, temp, etc) and an assignment operator
// case 2:
// tac[7] = "function";
```

```
// break;
//CASE for writeln
case 3:
   num_filled++;
   tac[num_filled-1] = "writeln";
   for(i = 0; i < num_filled -2; i++){
      fprintf(fout, "%s ", tac[i].c_str());
   }
   fprintf(fout, "%s\n",tac[i].c_str());

break;
//CASE for write
case 4:
   num_filled++;
   tac[num_filled-1] = "write";
   for(i = 0; i < num_filled -2; i++){
      fprintf(fout, "%s ", tac[i].c_str());
   }
   fprintf(fout, "%s\n", tac[i].c_str());

break;
//CASE for readln
case 5:
   num_filled++;
   tac[num_filled-1] = "readln";
   for(i = 0; i < num_filled -2; i++){
      if(i == 0){
         fprintf(fout, "%s := ", tac[i].c_str());
      }
```

```
        else{
            fprintf(fout, "%s ", tac[i].c_str());
        }
    }
    fprintf(fout, "%s new_line\n", tac[i].c_str());
    break;
//CASE for read
case 6:
    num_filled++;
    tac[num_filled-1] = "read";
    for(i = 0; i < num_filled -2; i++){
        if(i == 0){
            fprintf(fout, "%s := ", tac[i].c_str());
        }
        else{
            fprintf(fout, "%s ", tac[i].c_str());
        }
    }
    fprintf(fout, "%s\n", tac[i].c_str());
break;
//Eexpression & Assignment
case 7:
    num_filled++;
    tac[num_filled-1] = "expression";
    for(i = 0; i <num_filled-2 ; i++){
        fprintf(fout, "%s ",tac[i].c_str());
    }
    fprintf(fout, "%s\n", tac[i].c_str());
break;
```

```
        //Write "begin" and "end" with the scope they are related
        case 8:
        case 9:
            printf("%s %s\n", tac[0].c_str(), tac[1].c_str());
            fprintf(fout, "%s %s\n", tac[0].c_str(), tac[1].c_str());
        break;
        default:
            return;

    }
    return;
}
void add_3_ac_node(string args[7], int num_filled){
    three_ac_node * temp = new three_ac_node;
    temp->id = args[num_filled-1];
    temp->slots_used = num_filled;
    temp->next = NULL;
    if(args[num_filled].compare("procedure") == 0){
        temp->params = find_param_list(args[2]);
    }
    else{
        temp->params = NULL;
    }
    if(three_ac_list_head == NULL){
        three_ac_list_head = temp;
        three_ac_list_tail = temp;
    }
    else{
        three_ac_list_tail->next = temp;
```

```
        three_ac_list_tail = three_ac_list_tail->next;
    }
    return;
}
param_list_node * find_param_list(string label){
    param_list_node * current = param_list_head;
    if(current->label.compare(label.c_str()) == 0){
        //  cout << "found param list" << endl;
        //  cout << "list : " << current->label << endl;
        return current;
    }
    while(current->next != NULL){
        current = current->next;
        if(current->label.compare(label.c_str()) == 0){
            // cout << "found param list" << endl;
            return current;
        }
    }

    return NULL;
}
void print_list_of_param_lists(){
    param_list_node * current = param_list_head;
    while(current->next != NULL){
        cout <<"current->label: " << current->label << endl;
        current = current->next;
    }
    // cout << "current -> label :" << current->label << endl;
    // cout << "current -> tail :" << param_list_tail->label << endl;
```

```
    return;
}
string get_param_list_label(){
    param_list_node_count++;
    // cout << "param list node count : " << param_list_node_count << endl;
    param_list_node * temp = new param_list_node;
    temp->label = "PL" + to_string(param_list_node_count);
    temp->next = NULL;
    temp->list_head = NULL;
    temp->list_tail = NULL;

    if(param_list_head == NULL){
        // cout << "HEAD IS NULL" << endl;
        param_list_head = temp;
        param_list_tail = temp;
    }
    else{
        // cout << "HEAD IS NOT NULL" << endl;
        param_list_tail->next = temp;
        param_list_tail = param_list_tail->next;
    }
    return param_list_tail->label;
}
void add_param_to_list(param_list_node * params, string id, string type, int param_num){
    param_node * temp = new param_node;
    temp->next = NULL;
    temp->param_id = id;
    temp->param_type = type;
    temp->param_num = param_num;
```

```
   // cout << "params : " << params->label << endl;
   //  cout << "ID : " << id << " param num : " << param_num << endl;
   if(params->list_head == NULL){
      params->list_head = temp;
      params->list_tail = temp;
   }
   else{
      params->list_tail->next = temp;
      params->list_tail =  params->list_tail->next;
   }
   return;
}
void edit_param_type(param_list_node *params, int param_num, string type){

}
string trim_string(string string)
 {
   size_t pos = string.find_first_not_of(" ");
   string.erase(0, pos);

   pos = string.find_last_not_of(" ");
   if (string::npos != pos)
     string.erase(pos+1);
          return string;
 }

LINES = 2016
```