```
//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 <unordered_map>
#include <map>
#include <bits/stdc++.h>
#include <tuple>
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;

vector<sym_table_entry_t> sym_table;
string current_scope;
string current_scope_id;
stack<string> scope_stack;
int  current_temp_var_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 pr
ogram
*/
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 cont
aining an identifier and the params
   // the value will be of params in a non "null" situation will be a string for a temp
orary 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 s
tatements 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 file 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 * file, vector<icg_symbol>::iterat
or 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_file(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 * file, vector<icg_symbol>::iterator it, vector<icg
_symbol> icg_sym_table);
void print_sym_table();
int handle_io(FILE * file, vector<icg_symbol>::iterator it, vector<icg_symbol> icg
_sym_table);
int lookup(string search_val, int);
int set_var_value(string id, string value_ptr);
string get_temp_var();
//////////////////////////
```

```
void print_sym_table(){
    for(auto& tuple: sym_table){
        cout << "ID : " << get<0>(tuple) << " SCOPE : " << get<1>(tuple)<< " SCOP
E ID: "<<get<2>(tuple) << " VALUE : "
        << get<3>(tuple) << " TYPE: " << get<4>(tuple) << endl;
    }
    return;
}
/*
    Procedure: prases the symbol table and generates three address code for valid s
tatements
                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 * file = fopen("output_icg", "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, file, 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, file, 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, file, it, icg_sym_table);
            if(offset == -1){exit(-1);}
            it += offset-1;
        }
        else if(t_type.compare("begin_sym")==0){
            current_scope_id = scope_stack.top();
            cout << "Current Scope : " << current_scope_id << endl;

        }
        else if(t_type.compare("end_sym") == 0){
            string temp_s = scope_stack.top();
            cout << "Leaving Scope :" << current_scope_id << endl;
            scope_stack.pop();
            if(!scope_stack.empty()){
                current_scope_id = scope_stack.top();
            }
            //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(file, it, icg_sym_table);
            it += offset-1;
          }
          /*
             Handle Simplified expressions
          */


          /*
             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(file, it_prev, it_next, icg_sym_table);
          }

          /*
             Handle read/wriite I/O calls
          */
          else if((t_type.compare("write_sym") == 0) || (t_type.compare("read_sym")
== 0)){
             offset = handle_io(file, it, icg_sym_table);
             it += offset-1;
          }
          else if((t_type.compare("writeln_sym") == 0) || (t_type.compare("readln_sy
m") == 0)){
             offset = handle_io(file, 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("illegal") == 0){
             cout << "ERROR : unrecognized token" << endl;
          }
          else{
```

```
            // cout << "ELSE " << temp.token_type << " : " << temp.value << endl
;
        }
    }
    fclose(file);
}
int handle_io(FILE * file, 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.compar
e("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){
                    // cout << "writing tac to file for write|writeln " <<endl;
                    write_three_ac_to_file(file, tac, num_filled, flag);
                }
                else{
                    cout << "ERROR : expected ';' after statement : actual"<< sym.toke
n_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.com
pare("litchar") == 0)){
            it2++;offset++;
        }
    }
  }
  //case for handling read or readln
  else if((sym.token_type.compare("readln_sym") == 0) || (sym.token_type.compa
re("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.comp
are("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 file for read|readln " <<endl;
                        write_three_ac_to_file(file, tac, num_filled, flag);

                    }
                }
            }
            else if((sym.token_type.compare("identifier") == 0) || (sym.token_type.c
ompare("litchar") == 0)){
                it2++;offset++;
            }
        }
    }
    else{
        cout << "ERROR: invalid token" << endl;
        it2++; offset++;
    }
    cout << "-------------------------------------------------"<<endl;
    return offset;
```

```
}
int handle_var_declaration(FILE * file, vector<icg_symbol>::iterator it, vector<icg
_symbol> icg_sym_table){
    cout << "-------------------------------------------------"<<endl;
    cout << "       In var delcaration for "<< current_scope_id << 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 << "breaking here" << 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("litchar") == 0) || (sym.token_type.compare(
"identifier") == 0)){
            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        val
ue      type
```

```
                    sym_table_entry_t value_temp = make_tuple(temp_id, current_scop
e, current_scope_id, "undefined", "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("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 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 temp_var = "";
    string prev_temp_var = "";
    bool temp_set = false;
    bool prev_temp_set = false;
    stack <string > s;
    // cout << "sym val: " << sym.value << endl;
    if(lookup(sym.value, 1) < 0 ){
        cout << "ERROR : undefined reference to : " << sym.value << endl;
        // exit(-1);
    }
    else{
        cout << sym.value << " ";
        current_id = sym.value;
        it2++;
        sym = *it2;
        while(sym.token_type.compare("semicolon")!=0){
            // cout << sym.value << " ";
            s.push(sym.value);
            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;
            }
            // 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;
        }
        // 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(("t"+to_string(current_temp_var_num)));
            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 << "op_count : " << op_count << endl;

                //skip what we will handle later which is assigning the last temp variable
to the identifier
                if((sym.value.compare(current_id)==0) ||(sym.value.compare(":=") == 0)
){
                    it2++;offset++;
                    sym = *it2;
                }
                else if((sym.token_type.compare("number") == 0) || (sym.token_type.co
mpare("litchar") == 0)
                    || sym.token_type.compare("identifier")== 0){
                    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.co
mpare("litchar") == 0)
                        || sym.token_type.compare("identifier")== 0){
                        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;
                            string 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_file(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 if(sym.token_type.compare("lparen") == 0){
```

```
                // cout << "here" << endl;
                it2++;offset++;
                sym = *it2;
                // while((sym.token_type.compare("rparen") != 0) && (op_count >0))
{
                    if((sym.token_type.compare("number") == 0) || (sym.token_type.co
mpare("litchar") == 0)
                     || (sym.token_type.compare("identifier")== 0) || (temp_set == true
)){
                      set_value += sym.value; set_value += " ";
                        if(temp_set){
                            set_value.clear();
                            set_value += temp_var;
                        }

                     // 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_ty
pe.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;
                                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_file(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.comp
are("litchar") == 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;
                    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_file(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 << " setval : " << temp_var << endl;
    set_var_value(current_id , temp_var);
    tac[0] = current_id;
    tac[1] = ":=";
    tac[2] = set_value;
    write_three_ac_to_file(fout, tac, 3, 7);
}
else{
```

```
            // cout << "setval : " << set_value << endl;
            set_var_value(current_id , set_value);
            tac[0] = current_id;
            tac[1] = ":=";
            tac[2] = set_value;
            write_three_ac_to_file(fout, tac, 3, 7);
        }

        cout << 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.e
nd(); ++it){
        sym_table_entry_t temp1 = *it;
        string ref;
        if ((get<1>(temp1).compare("temp") == 0) && get<3>(temp1).compare("und
efined") == 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.e
nd(); ++it){
        sym_table_entry_t temp1 = *it;
        sym_table_entry_t temp2;
        if ((get<0>(temp1).compare(id) == 0)){
            tie(idd,scope, scope_id, val, type) = temp1;
            val = value_ptr;
            temp2 = make_tuple(id,scope,scope_id,val,type);
            // cout << "New Val : " <<get<3>(temp2) << endl;
```

```
            replace(sym_table.begin(), sym_table.end(), temp1, temp2);
            return 0;
        }
    }
    return 0;
}
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_ta
ble.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_ta
ble.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, "undefin
ed", "integer");
    cout << "Adding " << id << " to symbol table" << endl;
    sym_table.push_back(entry);
    return 0;
  }
  //check if its in the scope;
  return -1;
}
int gen_three_ac_prog_proc_func(int flag, FILE * fout, vector<icg_symbol>::iterat
or 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 file
  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 be
ing 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 litchar
s 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, curr
ent_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, curr
ent_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, curren
t_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_file(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 litch
ars 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, c
urrent_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, c
urrent_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, cur
rent_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_file(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 file 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 new_line\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 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 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;
    //EXPRESSIONs
    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;
        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){

}
```