Parser.y

```
%{

#include <stdio.h>

#include <stdlib.h>

#include <bits/stdc++.h>

#include <fstream>

#include "helper1.cpp"


using namespace std;


int yylex(void);

void yyerror(char *s){

   fprintf(stderr,"Unknown errors detected.\n");

   extern int lineno;

   }




extern int lineno;

extern int lineno;


vector<function_struct> func_table;

struct function_struct *active_func_ptr;

struct function_struct call_name_ptr;

vector<val_type> args_list;

vector<variable> var_list;




string global_func_name = "Global";

struct function_struct global(global_func_name ,NONE);


int sem_flag=0,gen_flag=0,level=0,isvar=1;
```

```
%}


%union{

	node *Node;

}


%token		<Node>NUM INT FLOAT FLOAT_CONST STRING STRING_CONST BOOL DEL ID
COMPARATOR CLOSESQ OPENSQ OPENBR BOOL_CONST

%token  <Node>  CLOSEBR OPENPR PROCESSORS ID1

%token  <Node>  CLOSEPR EQUAL QUOTES COMMA COLON DOT

%token		<Node>FOR WHILE IF ELSE

%token  <Node>  IS_RUNNING SUBMIT_JOBS GET_CLOCK_SPEED DISCARD_JOB JOB_ID

%token  <Node>  GET_AVAILABLE_MEMORY GET_MEMORY MEM_SIZE

%token  <Node>  PROCESSOR ISA CLOCK_SPEED L1_MEM ARM AMD CDC MIPS MEMORY
MEMORY_TYPE MEMORY_SIZE

%token  <Node>  PRIMARY SECONDARY CACHE LINK START_POINT END_POINT BANDWIDTH JOB
FLOPS_REQUIRED

%token  <Node>  DEADLINE MEM_REQUIRED AFFINITY RUN WAIT

%token  <Node>  CLUSTER TOPOLOGY NAME STAR RING BUS

%token  <Node>  SUM VOID

%token <Node>MULT RETURN

%token <Node>UNARY_OP

%start   program


%type <Node> program statement_list statement var_decl loop ifstmnt var_type variable for_loop
while_loop expr condition arithmatic_op mul factor func_dec func_head result_id return_type
decl_plist decl_list return_statement return_value function_call func_call parameter_list param
parameter array constants


%%

program : statement_list {


	printf("program : statement_list\n\n\n");
```

```
            $$ = add_node("program",$1);
   //$$->code = $1->code;
   //$$->code = generate_final_code($$->code,global_var_table);
   if( gen_flag==0 && sem_flag==0){
      printf("Compilation successful.\n");
      print_nodes($$);
      //ofstream output;
             //output.open("tree.txt");
      //print_nodes($$,output);
      //print_symbol(global_var_table);
      //print_func(func_table);
      //output.close();
      //fprintf(fout,$$->code.c_str());
      //printf("ASM file generated.\n");
   }
   else if( gen_flag==1 && sem_flag==0){
      printf("Syntax errors found.\n");
   }
   else if(gen_flag==0 && sem_flag==1){
      printf("Semantic errors found.\n");
   }
   print_func(func_table);


}
                        ;


statement_list  : statement statement_list {
                        printf("statement_list : statement statement_list\n");
                        $$ = add_node("statement_list",$1,$2);
               //$$->code = $1->code + $2->code;
   }
```

```
          | {
                    printf(" matched epsilon\n");
                    $$ = NULL;
          }
          ;

statement : var_decl DEL{
                    printf("statement : var_decl DEL\n");
                    $$ = add_node("statement",$1,$2);
                    //$$->code = $1->code + $2->code;
                    }
          | loop {
                    printf("statement : loop\n");
                    $$ = add_node("statement",$1);
                    //$$->code = $1->code;
                    }
          | ifstmnt {
                    printf("statement : ifstmnt \n");
                    $$ = add_node("statement",$1);
                    //$$->code = $1->code;
                    }
          | expr DEL {
                    printf("statement : expr DEL\n");
                    $$ = add_node("statement",$1,$2);
                    //$$->code = $1->code + $2->code;
                    }
          | func_dec {
                    printf("statement : func_dec \n");
                    $$ = add_node("statement",$1);
                    //$$->code = $1->code;
                    }
```

```
            | function_call DEL {
                        printf("statement : func_call \n");
                        $$ = add_node("statement",$1,$2);
                        //$$->code = $1->code + $2->code;
                        }
            ;

func_dec : func_head OPENBR statement_list return_statement CLOSEBR {
            printf("func_dec : func_head {statement_list} \n");
            }
                        ;

func_head : result_id OPENPR decl_plist CLOSEPR {
printf("func_head : result_id ( decl_plist ) \n");
                                    level=level+1;
                                    }
                                    ;

result_id : return_type ID1 {
                                    printf("result_id : return_type ID \n");
                                    if( check_func_name(func_table,$2->name)){
                                                printf("1\n");
                                                function_struct cur_func($2->name ,$1->type);
                                func_table.push_back(cur_func);
                                active_func_ptr = &func_table.back();
                                cout<<"Active function: "<<active_func_ptr->name<< " type: "
<<active_func_ptr->return_type<<endl;
                              }
                              else{
                                    printf("2\n");
                                       cout<<"Function name: "<<$2->name<<" already exists"<<endl;
```

```
                            $$->type=ERROR;
                            sem_flag=1;
                        }
                        isvar=0;
                      }
                      ;

decl_plist : decl_list  {
                          printf("decl_plist : decl_list \n");
                          $$ = add_node("decl_plist",$1);
                          isvar=1;
                          }
                          | {
                          printf("EPSILON IN decl_plist \n");
                          isvar=1;
                          }
                          ;

decl_list : var_decl COMMA decl_list {
                          printf("decl_list : var_decl COMMA decl_list \n");
                          //checked automatically in var decl of repitition
                          }
                          | var_decl {printf("decl_list : var_decl \n");}
                          ;

return_type : VOID {
                          printf("return_type : VOID \n");
                          $$->type=NONE;
                          }
                          | var_type {
                          printf("return_type : var_type \n");
```

```
                              $$->type=$1->type;
                              }
                              ;

return_statement: RETURN return_value DEL {
                                                  printf("return_statement: RETURN return_value DEL \n");
                                                  level = level-1;
                                                  cout<<"Active func return type: "<<active_func_ptr-
>return_type<<" variable type: "<<$2->type<<endl;
                                                  if(active_func_ptr->return_type!=$2->type){
                                                            sem_flag=1;
                                                            $$->type=ERROR;
                                                            cout<<"ERROR: Return type mismatch of function
"<<active_func_ptr->name<<endl;
                                                  }
                                                  active_func_ptr=&global;

                                        }
                                        | {
                                                  printf("MATCHED EPSILON IN RETURN STMNT \n");
                                                  level = level-1;
                                                  cout<<"Active func return type: "<<active_func_ptr-
>return_type<<" variable type: NONE"<<endl;
                                                  /*if(active_func_ptr->return_type!=NONE){
                                                                      sem_flag=1;
                                                                      $$->type=ERROR;


                                                                      printf("no return expected for function %s
at line no %d",active_func_ptr->name,lineno);
                                                            }*/
                                                  active_func_ptr=&global;
                                        }
                                        ;

return_value: variable {
                              printf("return_value: variable \n");
                              $$->type=$1->type;
                              }
                              | constants {
                              printf("return_value: constants \n");
                              $$->type=$1->type;
                              }
                              ;

function_call: ID EQUAL func_call {
                                        struct variable *temp=find_var(active_func_ptr,&global,$1->name);
                                        if( temp!=NULL ){
                                                  $1->type = temp->type;

                                                  if($1->type !=$3->type){
                                                            cout<<"ERROR: Type mismatch for function
call"<<endl;
                                                            sem_flag = 1;
                                                  }
                                                  else{

                                                  }
                                        }
                                        else{
                                                  $$->type=ERROR;
                                                  cout<<"ERROR: Variable "<< $1->name<<" not
declared"<<endl;
                                                  sem_flag=1;
```

```
                    }
                    printf("function_call: ID EQUAL func_call \n");
                    }
                | func_call {printf("function_call: func_call \n");}
                ;

func_call : ID1 OPENPR parameter_list CLOSEPR {
                printf("func_call : ID1 OPENPR parameter_list CLOSEPR\n");
                function_struct * temp = find_function(func_table,$2->name);

                if(!temp ){
                    $$->type = temp->return_type;

                }
                else{
                    printf("Error at line %d: Function not declared.\n",lineno);
                    sem_flag = 1;
                }
                }

parameter_list: param {printf("parameter_list: param \n");}
                |   {printf("parameter_list: epsilon \n");}
                ;

param: param COMMA parameter {
            printf("param: param COMMA parameter \n");

            }
            | parameter {
            printf("param: parameter \n");
            }
```

```
                    ;

parameter : variable {
                    printf("parameter : variable \n");
                    $$->type=$1->type;
                    $$->val=$1->val;
                    add_node("parameter",$1);
                    }
                    | constants {
                    printf("parameter : constants \n");
                    $$->type=$1->type;
                    $$->val=$1->val;
                    add_node("parameter",$1);
                    }
                    ;

constants : NUM  {
                    printf("constants : NUM\n");
                    $$->type=$1->type;
                    $$->val=$1->val;
                    add_node("constants",$1);
                    }
                    | FLOAT_CONST {
                    printf("parameter_list: FLOAT_CONST\n");
                    $$->type=$1->type;
                    $$->val=$1->val;
                    add_node("constants",$1);
                    }
                    | STRING_CONST {
                    printf("parameter_list: STRING_CONST\n");
                    $$->type=$1->type;
```

```
                $$->val=$1->val;
                add_node("constants",$1);
                }
                | BOOL_CONST {
                printf("parameter_list: BOOL_CONST\n");
                $$->type=$1->type;
                $$->val=$1->val;
                add_node("constants",$1);
                }


                ;


var_decl : var_type ID {
        printf("var_decl : var_type variable\n");
        if(isvar==0){
                printf("the given id is a parameter\n");
                if(check_func_varlist(active_func_ptr,$2->name)){
                        variable newvar=variable($2->name,$1->type,$2->type1,level);
                        newvar.dim=$2->dim;
                        active_func_ptr->params.push_back(newvar);
                }
                else{
                 $$->type=ERROR;
                 printf("ERROR: Redeclaration of parameter ");
                 cout<<$2->name<<endl;
                 sem_flag=1;
                }
        }
        else{
                printf("The given id is a variable\n");
                if(check_func_varlist(active_func_ptr,$2->name)){
```

```
                                variable newvar=variable($2->name,$1->type,$2->type1,level);
                                newvar.dim=$2->dim;
                                active_func_ptr->local_var.push_back(newvar);
                        }
                        else{
                         $$->type=ERROR;
                         printf("ERROR: Redeclaration of variable ");
                         cout<<$2->name<<endl;
                         sem_flag=1;
                        }
                }
        }
                         ;

var_type : INT{
                        printf("var_type : INT\n");
                        $$ = add_node("var_type",$1);
                 $$->type = INT1;
                 printf("$$-type = int\n");
                 }
                | STRING{
                        printf("var_type : STRING\n");
                        $$ = add_node("var_type",$1);
                 $$->type = STRING1;
                 }
                | BOOL{
                        printf("var_type : BOOL\n");
                        $$ = add_node("var_type",$1);
                 $$->type = BOOL1;
                 }
                | FLOAT{
                    printf("var_type : FLOAT\n");
                    $$ = add_node("var_type",$1);
                    $$->type = FLOAT1;
                    }
          ;

variable: array {
                              printf("variable : ARRAY \n");
                              $$->type1=ARRAY1;
                              $$->name=$1->name;
                              $$->dim=$1->dim;
                    }
                    | ID  {
                              printf("variable : ID \n");
                              $$->name=$1->name;
                              struct variable *temp=find_var(active_func_ptr,&global,$1->name);
                              if( temp!=NULL ){
                                        $$->val=temp->val;
                                        $$->type = temp->type;
                              }
                              else{
                                        $$->type=ERROR;
                                        printf("ERROR: variable in use is not declared at line no
%d\n",lineno);
                                        sem_flag=1;
                              }
                    }
                    ;

array: ID OPENSQ NUM CLOSESQ{
                    printf("array: ID OPENSQ NUM CLOSESQ \n");
                    $$->dim=$3->val;
                    $$->name=$1->name;
                    $$->type1=ARRAY1;
          }
          ;

loop : for_loop{printf("loop : for_loop\n");}
           | while_loop{printf("loop : while_loop\n");}
           ;

for_loop : FOR OPENPR expr DEL condition DEL expr CLOSEPR OPENBR statement_list CLOSEBR
{printf("for_loop\n");}

while_loop : WHILE OPENPR condition CLOSEPR OPENBR statement_list CLOSEBR
{printf("while_loop\n");}

ifstmnt : IF OPENPR condition CLOSEPR OPENBR statement_list CLOSEBR ELSE OPENBR
statement_list CLOSEBR {printf("ifstmnt\n");}
                    | IF condition OPENBR statement_list CLOSEBR {printf("ifstmnt\n");}


expr : ID EQUAL condition {
                    printf("expr : ID EQUAL condition\n");
                    $$ = add_node("expr",$1,$2,$3);
                    struct variable *temp=find_var(active_func_ptr,&global,$1->name);
                    if( temp!=NULL ){
                              temp->val=$3->val;
                    }
                    else{
                              $$->type=ERROR;
                              printf("variable in use is not declared, error at line no %d\n",lineno);
                              sem_flag=1;
```

```
                }
            }
            | ID EQUAL  arithmatic_op {
                $$ = add_node("expr",$1,$2,$3);
        //print_nodes($$);
                printf("expr : ID EQUAL  arithmatic_op\n");
                /*struct variable *temp=find_var(active_func_ptr,&global,$1->name);
                if( temp!=NULL ){
                    temp->val=$3->val;
                }
                else{
                    $$->type=ERROR;
                    printf("variable in use is not declared, error at line no %d\n",lineno);
                    sem_flag=1;
                }*/
            }
            ;

condition : arithmatic_op COMPARATOR arithmatic_op {
                    printf("condition : arithmatic_op COMPARATOR arithmatic_op\n");
                    $$ = add_node("condition",$1,$2,$3);
                    if( ($1->type==INT1 || $1->type==FLOAT1) && ($3->type==INT1 || $3-
>type==FLOAT1) ){
                        $$->type = BOOL1;
                    }
         else if ($1->type ==BOOL1 && $3->type==BOOL1 && ($2->name=="==" || $2->name=="!=") )
                $$->type = BOOL1;
           else
                $$->type = ERROR;

                    if($$->type == ERROR){
                        sem_flag=1;
                        cout<<"type mismatch at line no: "<<lineno<<endl;
                }
                else{
                        if( $2->name==">" )
                                $$->val =( $1->val > $3->val);
                        else if( $2->name==">=" )
                                $$->val =( $1->val >= $3->val);
                        else if( $2->name=="<" )
                                $$->val = ($1->val < $3->val);
                        else if( $2->name=="<=" )
                                $$->val =( $1->val <= $3->val);
                        else if( $2->name=="==" )
                                $$->val = ($1->val == $3->val);
                        else if( $2->name=="!=" )
                                $$->val = ($1->val != $3->val);
                }
                printf("$$->value : %f\n", $$->val);
        }
        | UNARY_OP factor {
                printf("condition : UNARY_OP factor\n");
                $$ = add_node("condition",$1,$2);
                if($2->type!=STRING1){
                        if($2->type!=BOOL1){
                                if($1->name=="!")
                                        $$->val=!$2->val;
                                else if($1->name=="++"){
                                        $2->val=$2->val+1;
                                        $$->val=$2->val;
                                }
                                else if($1->name=="--"){
                                        $2->val=$2->val-1;
                                        $$->val=$2->val;
                                    }
                                }
                                else if($1->name == "!")
                                    $$->val=!$2->val;
                                else{
                                    sem_flag=1;
                                    cout<<"unsuported operand with bool at line no: 
"<<lineno<<endl;
                                    $$->type=ERROR;
                                }
                            }
                            else{
                                sem_flag=1;
                                cout<<"unsuported operand with string at line no: "<<lineno<<endl;
                                $$->type=ERROR;
                            }
                            printf("$$->value : %f\n", $$->val);
                        }
                        | factor UNARY_OP {
                            printf("condition : arithmatic_op UNARY_OP\n");
                            $$ = add_node("condition",$1,$2);
                                if($1->type!=STRING1){
                                    if($1->type!=BOOL1){
                                        if($2->name=="!")
                                            $$->val=!$1->val;
                                        else if($2->name=="++"){
                                            $1->val=$1->val+1;
                                            $$->val=$1->val;
                                        }
                                        else if($2->name=="--"){
                                                $1->val=$1->val-1;
                                                $$->val=$1->val;
                                        }
                                }
                                else if($2->name == "!")
                                        $$->val=!$1->val;
                                else{
                                        sem_flag=1;
                                        cout<<"unsuported operand with bool at line no: 
"<<lineno<<endl;
                                        $$->type=ERROR;
                                }
                        }
                        else{
                                sem_flag=1;
                                cout<<"unsuported operand with string at line no: 
"<<lineno<<endl;
                                $$->type=ERROR;
                        }
                        printf("$$->value : %f\n", $$->val);
                }
                ;

arithmatic_op : mul SUM arithmatic_op {
                        printf("mul : mul SUM arithmatic_op\n");
                     $$ = add_node("arithmatic_op",$1,$2,$3);
                        $$->type=coercible($1->type,$3->type);
                        cout<<"types "<<$$->type<<" "<<$1->type<<" "<<$3->type<<endl;
                        if($$->type == ERROR){
                                sem_flag=1;
                                cout<<"type mismatch at yyline no: "<<lineno<<endl;
                        }
                        else{
                                if( $2->name=="+" )
                                        $$->val = $1->val + $3->val;
                                else
                                        $$->val = $1->val - $3->val;
                        }
                        printf("$$->value : %f\n", $$->val);
                }
          | mul {
                printf("arithmatic_op : mul\n");
                $$ = add_node("arithmatic_op",$1);
                        $$->val = $1->val;
                        $$->type = $1->type;
                        cout<<"types "<<$$->type<<" "<<$1->type<<endl;
                        printf("$$->val: %f\n", $$->val);
        //print_nodes($$);



          }
        ;

mul : factor MULT mul {
        printf("mul : factor MULT mul\n");
     $$ = add_node("mul",$1,$2,$3);
        $$->type=coercible($1->type,$3->type);
        cout<<"types "<<$$->type<<" "<<$1->type<<" "<<$3->type<<endl;

        if($$->type == ERROR){
                sem_flag=1;
                cout<<"type mismatch at yyline no: "<<lineno<<endl;
        }
        else{
                if( $2->name=="*" )
                        $$->val = $1->val * $3->val;
                else{
                        if($3->val==0){
                                sem_flag=1;
                                $$->type=ERROR;
                                cout<<"error division by 0 at line no: "<<lineno<<endl;
                        }
                        else
                                $$->val = $1->val / $3->val;
                }
        }
}

| factor {
     $$ = add_node("mul",$1);
        printf("mul : factor\n");
        $$->val = $1->val;
        $$->type = $1->type;
        cout<<"types "<<$$->type<<" "<<$1->type<<endl;
        printf("$$->val: %f\n", $$->val);


        }
;

factor : ID {
                printf("factor : ID \n");
        $$ = add_node("factor",$1);
                struct variable *temp=find_var(active_func_ptr,&global,$1->name);
                if( temp!=NULL ){
                        $$->val=temp->val;
                        $$->type = temp->type;
                }
                else{
                        $$->type=ERROR;
                        printf("variable in use is not declared, error at line no %d\n",lineno);
                        sem_flag=1;
                }

        }
| OPENPR arithmatic_op OPENPR {
        $$ = add_node("factor",$1);
          printf("factor : OPENPR arithmatic_op OPENPR \n");
          $$->type = $2->type;
          $$->val = $2->val;
                }
| NUM {
          printf("factor : NUM \n");
        $$ = add_node("factor",$1);
          $$->type = INT1;
                cout<<"types "<<$$->type<<" "<<$1->type<<endl;
          $$->val=$1->val ;
                cout<<"val "<<$$->val<<" "<<$1->val<<endl;

          }
| FLOAT_CONST {
          printf("factor : FLOAT_CONST \n");
                $$ = add_node("factor",$1);
          $$->type = FLOAT1;
                $$->val=$1->val ;
```

```
                                    }
                | BOOL_CONST {
                            printf("factor : BOOL_CONST \n");
                   $$ = add_node("factor",$1);
                            $$->val=$1->val ;
                            $$->type = BOOL1;
                          }
                          | array{
                                    printf("factor : ID \n");
                         $$ = add_node("factor",$1);
                                    $$->type = $1->type;
                                    $$->type1 = $1->type1;
                            $$->val = $1->val;
                          }
                ;

%%
int main(){
            active_func_ptr = &global;
   func_table.push_back(global);
   yyparse();
   return 0 ;



}
Helper1.cpp
#include "struct.h"
#include <bits/stdc++.h>

using namespace std;
```

```
node* add_node(string name, node* a=NULL, node* b=NULL, node* c=NULL, node* d=NULL, node*
e=NULL, node* f=NULL, node* g=NULL, node* h=NULL, node* i=NULL, val_type type= NONE, int
val=0) {
    static int no = 1;


    node *new_node;
    new_node = new node();
    new_node->children[0] = a;
    new_node->children[1] = b;
    new_node->children[2] = c;
    new_node->children[3] = d;
    new_node->children[4] = e;
    new_node->children[5] = f;
    new_node->children[6] = g;
    new_node->children[7] = h;
    new_node->children[8] = i;
    new_node->children[9] = NULL;
    new_node->node_name=name;
    new_node->name;
    new_node->node_no = no * 10;
    new_node->type=type;
    new_node->val=val;


    return new_node;
}


val_type coercible(val_type expr1,val_type expr2){
    if(expr1==INT1 && expr2==INT1)
        return INT1;
    else if( (expr1==INT1 && expr2==FLOAT1) || (expr2==INT1 && expr1==FLOAT1) ||
(expr2==FLOAT1 && expr1==FLOAT1) )
        return FLOAT1;
    else
        return ERROR;
}

// val_type comparable(val_type expr1,val_type expr2){
//      if( (expr1==INT1 || expr1==FLOAT1) && (expr2==INT1 || expr2==FLOAT1)){
//          return BOOL1;
//      }
//          else{
//              sem_flag=1;
//              cout<<"unsuported operand with string at line no: "<<yylineno<<endl;
//              $$->type=ERROR;
//          }
// }

// void print_local_var(function *active_func_ptr)
// {
//      vector<variable>:: iterator it;
//      for(it = active_func_ptr->local_var.begin() ; it != active_func_ptr->local_var.end(); ++it)
//      {
//              cout << it->name << " " << it->type << "  " << it->ele_type  <<"\n";
//      }
// }

void print_nodes(struct node* root)
{
    if(root == NULL)
        return;
    else
    {
        cout << "Parent node " << root->node_no << "( " << root->node_name << " ) : ";
```

```
          // myfile << "Parent node " << root->node_no << "( " << root->node_name << " ) : ";
          if(root->children[0] == NULL)
          {
                 cout << root->name;
                 // myfile << root->name;
          }
          for(int i=0 ; i<10 ; i++)
          {
                 if(root->children[i] != NULL)
                 {
                        cout << root->children[i]->node_no << " (" << root->children[i]->node_name << ")
";
                         // myfile << root->children[i]->node_no << " (" << root->children[i]->node_name
<< ") ";
                 }
                 else
                 {
                        cout << "\n";
                        // myfile << "\n";
                        break;
                 }
          }
          for(int i=0;i<10; i++)
          {
                 if(root->children[i] != NULL)
                 {
                        print_nodes(root->children[i]);
                        //break;
                 }
          }
   }
```

```
}




// int check_varlist(vector<variable> var_list, vector<function> func_table, int level,string name1)
// {

//      vector<variable>:: iterator it;
//      for(it = var_list.begin() ; it != var_list.end(); ++it)
//      {
//           if(it->name == name1)
//           {
//                return 0;
//           }
//      }
//      for (size_t i = 0; i < func_table.size(); i++) {
//         if(func_table[i].name == name1)
//              return 0;
//      }
//      return 1;
// }

int check_func_varlist(function_struct *current,string var_name){
    vector<variable> var_list = current->local_var;
    vector<variable> param_list = current->params;
    for(int i=0; i < param_list.size();i++){
      if(param_list[i].name == var_name)
      return 0;
    }
```

```
     for( int i=0;i< var_list.size(); i++){
        if(var_list[i].name == var_name ){
                return 0;
           }
        }
    return 1 ;
}

struct variable *find_var(function_struct *current,function_struct *global,string var_name){
     vector<variable> var_list = current->local_var;
     vector<variable> param_list = current->params;
     for(int i=0; i < param_list.size();i++){
        if(param_list[i].name == var_name)
           return &param_list[i];
     }

     for( int i=0;i< var_list.size(); i++){
        if(var_list[i].name == var_name ){
           return &var_list[i];
        }
    }
     var_list = global->local_var;
     for( int i=0;i< var_list.size(); i++){
        if(var_list[i].name == var_name ){
          return &var_list[i];
        }
    }
    return NULL ;
}
```

```
// int check_all_varlist(function *current, int level, string var_name, vector<variable>
global_var_table)

// {


// }


int check_func_name(vector<function_struct> func_list, string name){

   for(int i=0;i< func_list.size() ; i++){

          if( func_list[i].name == name)

                 return 0;

   }

   return 1;

}


function_struct * find_function(vector<function_struct> func_list, string name){

            for(int i=0;i< func_list.size() ; i++){

          if( func_list[i].name == name)

                 return &func_list[i];

   }

   return NULL;

}




void print_func_varlist(function_struct *current){


    vector<variable> variable_list = current->local_var;

    vector<variable> param_list = current->params;

    for(int i=0; i < param_list.size();i++){

            cout<<"Parameter "<<i<<" is "<<param_list[i].name<< " type "<<param_list[i].type<<endl;
    }

    for( int i=0;i< variable_list.size(); i++){
            cout<<"Variable "<<i<<" is "<<variable_list[i].name<< " type "<<variable_list[i].type<<endl;
    }
}

void print_func(vector<function_struct> func_list){
       vector<function_struct>:: iterator it;
       for(it = func_list.begin() ; it != func_list.end(); ++it){
              cout << it->name << " " << it->return_type << "\n";
              vector<variable> variable_list = it->local_var;
              vector<variable> param_list = it->params;
              for( int i=0;i< variable_list.size(); i++){
                                                          cout<<"Variable "<<i<<" is "<<variable_list[i].name<< " type
"<<variable_list[i].type<<endl;              }
              for( int i=0;i< param_list.size(); i++){
                 cout<<"Parameter "<<i<<" is "<<param_list[i].name<< " type "<<param_list[i].type<<endl;
              }
       }
}
```

Helper.cpp

```
#include "struct.h"
#include <bits/stdc++.h>

using namespace std;

node *terminal_node(string name, string value, val_type type=NONE,var_type type1=SIMPLE){
   node *new_node;
   new_node = new node();
```

```
    new_node->node_name=name;
    new_node->name=value;
    new_node->type=type;
    new_node->type1=type1;

    if(name.compare("NUM") == 0) {
        new_node->val = atoi(value.c_str());
    }
    else if(name.compare("FLOAT_CONST") == 0) {
        new_node->val = atof(value.c_str());
    }
    else if(name.compare("STRING_COST") == 0) {
        new_node->val1 = value;
    }
    else if(name.compare("BOOL_CONST") == 0) {
        if(value.compare("true") == 0)
            new_node->val = 1;
        else
            new_node->val = 0;
    }
    else if(name.compare("ARRAY") == 0) {
        new_node->type1=ARRAY1;

    }

    return new_node;
}
```

Struct.h

```
#include <bits/stdc++.h>
```

```
#include <cstdio>

#include <cstring>


using namespace std;


enum val_type { INT1, FLOAT1, STRING1, BOOL1, NONE, ERROR };

enum var_type { SIMPLE , ARRAY1};


struct variable{


  string name;

  val_type type;

  var_type ele_type;

  int dim;

  int level;

  int offset;

  float val=0;


  variable() {};


  variable(string name1 ,val_type type1 ,var_type ele_type1 ,int level1 ){

      name = name1;

      type = type1;

      ele_type = ele_type1;

      level = level1;


  }

};


struct function_struct{

   string name;
```

```
    val_type return_type;
    vector<variable> params;
    vector<variable> local_var;

    function_struct(){};
    function_struct(string name1 ,val_type return_type1){
        name = name1;
        return_type = return_type1;
    }

};

struct node{
    string node_name;
    node* children[10];

    string name;
    int line_no;
    int node_no;
    float val;
    int dim;
    string val1;
    val_type type;
    var_type type1;
    string code;
};
```

Lexer.l

```
%{
#include "helper.cpp"
```

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

#include <stdlib.h>

char * xyz = "lkajdsflkjasdf";

int lineno = 1;

%}


%option yylineno

QUOTES                  "'"

QUOTES1         \"

DIGIT           [0-9]

STRING          [a-zA-Z0-9]+

TEXT_NUMBERS [a-zA-Z0-9]

NUM             {DIGIT}+

ID                      [a-z]{TEXT_NUMBERS}*

ID1                     [A-Z]{TEXT_NUMBERS}*

VAR1            {ID}"["{NUM}"]"

FLOATCONST      {NUM}"."{NUM}

SUM             "+"|"-"

MULT            "*"|"/"

LOGICAL_OP      "&"|"\|"

UNARY_OP        "!"|"++"|"--"

COMPARATOR  ">"|"<"|">="|"<="|"=="|"!="


%%

"("                     { yylval.Node = terminal_node("OPENPR","("  ); return OPENPR;}

")"                             { yylval.Node = terminal_node("CLOSEPR", ")" ); return CLOSEPR;
        }

"{"                             { yylval.Node = terminal_node("OPENBR", "{" );  return OPENBR;
        }

"}"                             { yylval.Node = terminal_node("CLOSEBR", "}" );  return CLOSEBR;
        }

"["                             { yylval.Node = terminal_node("OPENSQ", "[" ); return OPENSQ; }
"]"                         { yylval.Node = terminal_node("CLOSESQ", "]" ); return CLOSESQ;
        }
"."                         { yylval.Node = terminal_node("DOT", "." ); return DOT;  }
","                         { yylval.Node = terminal_node("COMMA", "," ); return COMMA; }
";"                         { yylval.Node = terminal_node("DEL", ";" ); return DEL;           }
":"                         { yylval.Node = terminal_node("COLON", ":" ); return COLON;     }
"="                         { yylval.Node = terminal_node("EQUAL", "=" ); return EQUAL;     }
{NUM}                 { yylval.Node = terminal_node("NUM", string(yytext), INT1 ); return NUM;
        }
{FLOATCONST}  { yylval.Node = terminal_node("FLOAT_CONST", string(yytext), FLOAT1 ); return
FLOAT_CONST;  }
{SUM}                 { yylval.Node = terminal_node("SUM", string(yytext) ); return SUM;
        }
{MULT}                { yylval.Node = terminal_node("MULT", string(yytext) ); return MULT;     }
{UNARY_OP}            { yylval.Node = terminal_node("UNARY_OP", string(yytext) ); return
UNARY_OP;             }
{COMPARATOR}          { yylval.Node = terminal_node("COMPARATOR", string(yytext) ); return
COMPARATOR;}
"true"        { yylval.Node = terminal_node("BOOL_CONST","true"); return BOOL_CONST; }
"false"       { yylval.Node = terminal_node("BOOL_CONST","false"); return BOOL_CONST; }
"if"                  { yylval.Node = terminal_node("IF", "if" ); return IF;              }
"while"               { yylval.Node = terminal_node("WHILE", "while" ); return WHILE;          }
"else"                { yylval.Node = terminal_node("ELSE", "else" ); return ELSE;      }
"for"                 { yylval.Node = terminal_node("FOR", "for" ); return FOR;                }
"int"                 { yylval.Node = terminal_node("INT", "int" ); return INT;          }
"float"               { yylval.Node = terminal_node("FLOAT", "float" ); return FLOAT;  }
"bool"                { yylval.Node = terminal_node("BOOL", "bool", BOOL1 ); return BOOL;     }
"string"          { yylval.Node = terminal_node("STRING", "string" ); return STRING;       }
"void"                { yylval.Node = terminal_node("VOID", "void" ); return VOID;      }
"return"              { yylval.Node = terminal_node("RETURN", "return" ); return RETURN;      }
{ID}                  { yylval.Node = terminal_node("ID", string(yytext) ); return ID;             }
{ID1}                 { yylval.Node = terminal_node("ID1", string(yytext) ); return ID1;
        }
```

```
\"{STRING}\"      { yylval.Node = terminal_node("STRING_CONST", string(yytext), STRING1 ); return
STRING_CONST;              }

'{STRING}'                 { yylval.Node = terminal_node("STRING_CONST", string(yytext), STRING1 );
return STRING_CONST;  }

{QUOTES}                   { yylval.Node = terminal_node("QUOTES", "'" ); return QUOTES;  }

{QUOTES1}                  { yylval.Node = terminal_node("QUOTES", "\"" ); return QUOTES;             }

%%


int yywrap (void) {return 1;}
```