

Appendix

Chantal Galvez and Kunal Mishra and Lixing Pan and Jing Zhang

May 5, 2012

1 Lexical Analyzer

```
/* author : Jing */

%{
#include <stdio.h>
#include "Parser.tab.h"

#ifdef _MYECHO
#define MYECHO myecho()
#else
#define MYECHO MYECHO
#endif
void myecho();
char * myTextCopy();
void countLine(const char* ptr);
#include "util.h"
#include "global.h"
%}

letter      [A-Za-z]
letter_     [A-Za-z_]
digit       [0-9]
floatconst  ({digit}*\.{digit}+|{digit}+\.{digit}*)([eE][+-]?{digit}+)?
intconst    {digit}+
identifier  ({letter}({letter_}|{digit}))*
strliteral  \"([^\\"\\]|\\.)*\"
comment     (\\/\\*(\\^\\*|\\(\\*+\\(\\^\\*\\/\\)))\\*+\\/\\)|(\\/\\/\\.*)

%%
"void"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return VOID; }
"bool"|"boolean" { MYECHO; yylval.LString.l = LEXLINECOUNTER; return BOOLEAN; }
"int"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return INTEGER; }
"float"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return FLOAT; }
"string"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return STRING; }
"vlist"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return VLIST; }
"elist"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ELIST; }
"vertex"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return VERTEX; }
"edge"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return EDGE; }
"graph"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return GRAPH; }

"func"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return FUNC_LITERAL; }
"if"        { MYECHO; yylval.LString.l = LEXLINECOUNTER; return IF; }
"else"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ELSE; }
"for"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return FOR; }
"foreach"   { MYECHO; yylval.LString.l = LEXLINECOUNTER; return FOREACH; }
"while"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return WHILE; }
"break"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return BREAK; }
"continue"  { MYECHO; yylval.LString.l = LEXLINECOUNTER; return CONTINUE; }
"return"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return RETURN; }

"outE"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return OUTCOMING_EDGES; }
"inE"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return INCOMING_EDGES; }
"strtV"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return STARTING_VERTICES; }
"endV"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ENDING_VERTICES; }
"allV"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ALL_VERTICES; }
"allE"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ALL_EDGES; }

"print"     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return PRINT; }
"length"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return LENGTH; }
```

```

"=="      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return EQ; }
"!=="     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return NE; }
"<="     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return LE; }
">="     { MYECHO; yylval.LString.l = LEXLINECOUNTER; return GE; }
"+="      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ADD_ASSIGN; }
"-="      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return SUB_ASSIGN; }
"*="      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return MUL_ASSIGN; }
"/="      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return DIV_ASSIGN; }
"||"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return OR; }
"&&"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return AND; }

"<:"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return APPEND; }
">:"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ARROW; }
"|"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return PIPE; }
"@@"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return AT; }

"{"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '{'; }
"}"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '}'; }
"("       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '('; }
")"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ')'; }
"["       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '['; }
"]"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ']'; }
"?"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '?'; }
";"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ';'; }
","       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ','; }
":"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ':'; }
"."       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '.'; }
"!"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '!'; }
"+"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ADD; }
"-"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return SUB; }
"*"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return MUL; }
"/"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return DIV; }
"="       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return '='; }
">"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return GT; }
"<"       { MYECHO; yylval.LString.l = LEXLINECOUNTER; return LT; }
"<<"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return LIN; }
">>"      { MYECHO; yylval.LString.l = LEXLINECOUNTER; return ROUT; }

"true"    { MYECHO; yylval.LString.l = LEXLINECOUNTER; return BOOL_TRUE; }
"false"   { MYECHO; yylval.LString.l = LEXLINECOUNTER; return BOOL_FALSE; }
{intconst} { MYECHO;
              yylval.LString.s = myTextCopy();
              yylval.LString.l = LEXLINECOUNTER;
              return INTEGER_CONSTANT;
            }

{floatconst} { MYECHO;
              yylval.LString.s = myTextCopy();
              yylval.LString.l = LEXLINECOUNTER;
              return FLOAT_CONSTANT;
            }

{identifier} { MYECHO;
              yylval.LString.s = myTextCopy();
              yylval.LString.l = LEXLINECOUNTER;
              return IDENTIFIER;
            }

{strliteral} { MYECHO;
              yylval.LString.s = myTextCopy();
              yylval.LString.l = LEXLINECOUNTER;
              return STRING_LITERAL;
            }

{comment}  { MYECHO; countLine(yytext); }

[ \t]      { }
[\n]       { LEXLINECOUNTER++; }
.          { errorInfo(ErrorUnrecognizedLexeme, LEXLINECOUNTER, "error :%d: unrecognized lexeme '%s'\n",
              LEXLINECOUNTER, yytext);
            }

%%

char * myTextCopy() {
    char * cpy = (char *) malloc (yyleng+1);
    strncpy( cpy, yytext, yylen );
    cpy[yyleng] = '\0';
    return cpy;
}

void countLine(const char* ptr) {
    int i;
    for (i=0; i<strlen(ptr); ++i) {
        if (ptr[i]=='\n') LEXLINECOUNTER++;
    }
}

```

```

    }
}

void myecho() {
    fprintf(yyout, "LEX:");
    fprintf(yyout, "YYLENG=%d:", yyleng);
    fprintf(yyout, "YYTEXT=\"");
    ECHO;
    fprintf(yyout, "\"\\n");
}

```

../src/LexAly.l

2 Parser

```

/*****
- Grammar Syntax :
  I/O : Chantal, Kunal
  Expression, Declaration, Auxiliary : Jing
  Statement : Lixing
- AST construction :
  I/O : Chantal, Kunal
  Expression, Declaration, Auxiliary : Jing
  Statement : Lixing
*****/

%{
#include <stdio.h>
#include <stdlib.h>

extern FILE *yyin;
extern void yyerror(const char *str);
extern int yywrap(void);
extern int yylex(void);
extern int yyparse(void);

#include "ASTree.h"
#include "SymbolTable.h"
#include "SymbolTableUtil.h"
#include "util.h"
#include "CodeGenUtil.h"
#include "CodeGen.h"
#include "global.h"
%}

/*****
 * General Options
 *****/
%error-verbose

/*****
 * Field names
 *****/
%union{
    struct Node*    node;
    struct {
        char *      s;
        long long   l;
    }LString;
    struct {
        int         i;
        long long   l;
    }LInteger;
}

// basic
%type <LInteger> assignment_operator unary_operator function_literal_type_specifier
%type <LString> IDENTIFIER STRING_LITERAL INTEGER_CONSTANT FLOAT_CONSTANT
%type <LString> '=' ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN APPEND
%type <LString> ADD SUB MUL DIV '!'
%type <LString> EQ NE LE GE LT GT OR AND LIN ROUT PRINT
%type <LString> ARROW PIPE AT
%type <LString> BOOL_TRUE BOOL_FALSE
%type <LString> OUTCOMING_EDGES INCOMING_EDGES STARTING_VERTICES ENDING_VERTICES
%type <LString> ALL_VERTICES ALL_EDGES
%type <LString> VOID BOOLEAN INTEGER FLOAT STRING VLIST ELIST VERTEX EDGE GRAPH
%type <LString> FUNC_LITERAL
%type <LString> IF ELSE FOR FOREACH WHILE BREAK CONTINUE RETURN MARK
%type <LString> '{' '}' '(' ')' '[' ']' ';' ',' ':' '.' '?' LENGTH

```

```

// declaration
%type <node> declaration
%type <node> basic_type_specifier declaration_specifiers
%type <node> init_declarator_list init_declarator simple_declarator
%type <node> parameter_list parameter_declaration
%type <node> initializer initializer_list

// expression
%type <node> expression assignment_expression logical_OR_expression
%type <node> logical_AND_expression equality_expression relational_expression
%type <node> additive_expression multiplicative_expression cast_expression
%type <node> unary_expression postfix_expression primary_expression
%type <node> graph_property pipe_property argument_expression_list argument_expression
%type <node> attribute constant

// statements
%type <node> start_nonterminal translation_unit
%type <node> external_statement statement
%type <node> expression_statement compound_statement selection_statement compound_statement_no_scope
%type <node> iteration_statement jump_statement declaration_statement
%type <node> statement_list foreach_declaration
%type <node> io_statement io_ext io_ext_list

// function
%type <node> function_definition
%type <node> function_literal_declaration
%type <node> function_head func_declarator func_id

// other
%type <node> scope_out
/*****
 *      TOKEN LIST
 *****/
/* TYPE RELATED */
%token VOID BOOLEAN INTEGER FLOAT STRING VLIST ELIST VERTEX EDGE GRAPH DYN_ATTRIBUTE
%token IDENTIFIER INTEGER_CONSTANT FLOAT_CONSTANT STRING_LITERAL
%token BOOL_TRUE BOOL_FALSE
/* FUNCTIONS RELATED */
%token FUNC_LITERAL
/* GRAPH RELATED */
%token OUTCOMING_EDGES INCOMING_EDGES STARTING_VERTICES ENDING_VERTICES
%token ALL_VERTICES ALL_EDGES
/* OPERATOR */
%token ADD SUB MUL DIV
%token OR AND
%token EQ NE
%token GT LT GE LE
%token ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN
%token APPEND ARROW PIPE AT MARK
%token BELONG
%token LIN ROUT PRINT LENGTH
/* CONTROL */
%token IF ELSE
%token FOR FOREACH WHILE
%token BREAK CONTINUE
%token RETURN
/* used in AST */
%token AST_TYPE_SPECIFIER AST_DECLARATION AST_COMMA
%token AST_ASSIGN AST_CAST
%token AST_UNARY_PLUS AST_UNARY_MINUS AST_UNARY_NOT
%token AST_FUNC_DECLARATOR AST_PARA_DECLARATION AST_FUNC

%token AST_INIT_ASSGN AST_LIST_INIT
%token AST_MATCH AST_ATTRIBUTE AST_GRAPH_PROP
%token AST_STAT_LIST AST_COMP_STAT AST_COMP_STAT_NO_SCOPE AST_EXT_STAT_COMMA

%token AST_IF_STAT AST_IFELSE_STAT
%token AST_WHILE AST_FOR AST_FOREACH
%token AST_JUMP_CONTINUE AST_JUMP_BREAK AST_JUMP_RETURN
%token AST_FUNC_CALL AST_ARG_EXPS AST_EXP_STAT
%token AST_ERROR AST_LIST_MEMBER
%token AST_PRINT AST_PRINT_COMMA AST_PRINT_STAT AST_READ_GRAPH AST_WRITE_GRAPH
%token AST_LENGTH AST_SCOPE_OUT
/*****
 *      PRECEDENCE & ASSOC
 *****/
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

/*****
 *      START SYMBOL
 *****/

```

```

*****/
%start start_nonterminal

%%

/*****
 *   BASIC CONCEPTS
 *****/
start_nonterminal
: translation_unit {
    $$ = $1;
    showASTandST($$, "Syntax + Semantic P1");
    if(!ERRNO) { // no syntax error, or declaration error
        char *mainBodyCode=NULL, *mainCode;
        char *funCode=NULL, *flCode=NULL;
        char *globalDecl=NULL;
        codeInclude(&globalDecl);
        codeIndentInit();
        codeAllGen($$, &mainBodyCode, &funCode);
        codeAllFuncLiteral($$, &flCode);
        codeAllGlobal($$, &globalDecl);
        mainCode = wapperMainCode(mainBodyCode);
        codeIndentFree();
        showASTandST($$, "Semantic P2 + Code Gen");
        if(!ERRNO){
            OUTFILESTREAM = fopen(OUTFILE, "w");
            if(globalDecl!=NULL) exportCode(globalDecl); // global
            if(flCode!=NULL) exportCode(flCode); // fl
            if(funCode!=NULL) exportCode(funCode); // func
            exportCode(mainCode); // main
            fclose(OUTFILESTREAM);
        }
        free(mainBodyCode);
        free(funCode);
        free(mainCode);
        free(globalDecl);
    }
    astFreeTree($$); // destroy AST
}
;

translation_unit
: external_statement { $$ = $1; }
| translation_unit external_statement {
    struct Node* leftNode = astLeftmostNode($1);
    long long ll = -1;
    if(leftNode!=NULL) ll = leftNode->line;
    $$ = astNewNode( AST_EXT_STAT_COMMA, 2, astAllChildren( 2, $1, $2 ), ll );
}
;

/*****
 *   STATEMENTS
 *****/
external_statement
: function_definition{
    $$ = $1;
}
| statement{
    $$ = $1;
}
;

statement
: expression_statement { $$ = $1; }
| compound_statement { $$ = $1; }
| selection_statement { $$ = $1; }
| iteration_statement { $$ = $1; }
| jump_statement { $$ = $1; }
| declaration_statement { $$ = $1; }
| io_statement { $$ = $1; }
;

expression_statement
: expression ';' {
    $$ = astNewNode( AST_EXP_STAT, 1, astAllChildren(1, $1), $1->line);
}
| ';' { $$ = astNewNode( AST_EXP_STAT, 0, NULL, $1.1); }
| expression error {
    astFreeTree($1); $$ = NULL;
}
;

```

```

;

statement_list
: statement { $$ = $1; }
| statement_list statement {
    struct Node* leftNode = astLeftmostNode($1);
    long long ll = -1;
    if(leftNode!=NULL) ll = leftNode->line;
    $$ = astNewNode( AST_STAT_LIST, 2, astAllChildren(2, $1, $2), ll );
}
;

compound_statement
: '{' ' ' {
    $$ = astNewNode( AST_COMP_STAT, 0, NULL, $1.1 );
}
| '{' scope_in statement_list scope_out '}' {
    $$ = astNewNode( AST_COMP_STAT, 2, astAllChildren(2, $3, $4), $1.1 );
}
| '{' error { $$ = NULL; }
| '{' scope_in statement_list scope_out error {
    astFreeTree($3);
    astFreeTree($4);
    $$ = NULL;
}
;

compound_statement_no_scope
: '{' ' ' {
    $$ = astNewNode( AST_COMP_STAT_NO_SCOPE, 0, NULL, $1.1 );
}
| '{' statement_list '}' {
    $$ = astNewNode( AST_COMP_STAT_NO_SCOPE, 1, astAllChildren(1, $2), $1.1 );
}
| '{' error { $$ = NULL; }
| '{' statement_list error {
    astFreeTree($2);
    $$ = NULL;
}
;

selection_statement
: IF '(' expression ')' compound_statement {
    $$ = astNewNode(AST_IF_STAT, 2, astAllChildren(2, $3, $5), $1.1 );
}
| %prec LOWER_THAN_ELSE ;
| IF '(' expression ')' compound_statement ELSE compound_statement {
    $$ = astNewNode(AST_IFELSE_STAT, 3, astAllChildren(3,$3, $5, $7), $1.1);
}
;

iteration_statement
: WHILE '(' expression ')' compound_statement {
    $$ = astNewNode(AST_WHILE, 2, astAllChildren(2, $3, $5), $1.1);
}
| FOR '(' expression ';' expression ';' expression ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, $3, $5, $7, $9), $1.1);
}
| FOR '(' expression ';' expression ';' ' ' ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, $3, $5, NULL, $8), $1.1);
}
| FOR '(' expression ';' ';' expression ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, $3, NULL, $6, $8), $1.1);
}
| FOR '(' expression ';' ';' ' ' ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, $3, NULL, NULL, $7), $1.1);
}
| FOR '(' ';' expression ';' expression ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, NULL, $4, $6, $8), $1.1);
}
| FOR '(' ';' expression ';' ' ' ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, NULL, $4, NULL, $7), $1.1);
}
| FOR '(' ';' ';' expression ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, NULL, NULL,$5, $7), $1.1);
}
| FOR '(' ';' ';' ' ' ')' compound_statement {
    $$ = astNewNode(AST_FOR, 4, astAllChildren(4, NULL, NULL, NULL, $6), $1.1);
}
| FOREACH '(' foreach_declaration ':' postfix_expression ')' compound_statement {
    $$ = astNewNode(AST_FOREACH, 3, astAllChildren(3, $3, $5, $7), $1.1);
}

```

```

    }
    ;

foreach_declaration
: basic_type_specifier IDENTIFIER {
    $$ = astNewNode(AST_DECLARATION, 2, astAllChildren(2, $1, astNewLeaf(IDENTIFIER, $2.s, $2.1)), $2.1);
    sTableDeclare($$);
}

jump_statement
: BREAK ';' { $$ = astNewNode(AST_JUMP_BREAK, 0, NULL, $1.1); }
| CONTINUE ';' { $$ = astNewNode(AST_JUMP_CONTINUE, 0, NULL, $1.1); }
| RETURN expression ';' { $$ = astNewNode(AST_JUMP_RETURN, 1, astAllChildren(1, $2), $1.1); }
| RETURN ';' { $$ = astNewNode(AST_JUMP_RETURN, 0, NULL, $1.1); }
| BREAK error { $$ = NULL; }
| CONTINUE error { $$ = NULL; }
| RETURN expression error { $$ = NULL; astFreeTree($2); }
| RETURN error { $$ = NULL; }
;

declaration_statement
: declaration { $$ = $1; }
| function_literal_declaration { $$ = $1; }
;

io_statement
: PRINT io_ext_list ';' {
    $$ = astNewNode(AST_PRINT_STAT, 1, astAllChildren(1, $2), $1.1);
}
| IDENTIFIER LIN IDENTIFIER ';' {
    {
        // FILE << Graph
        struct Node* tn1 = astNewLeaf(IDENTIFIER, $1.s, $1.1);
        struct Node* tn3 = astNewLeaf(IDENTIFIER, $3.s, $3.1);
        sTableLookupId(tn1);
        sTableLookupId(tn3);
        $$ = astNewNode(AST_WRITE_GRAPH, 2, astAllChildren(2, tn1, tn3), $2.1);
    }
}
| IDENTIFIER ROUT IDENTIFIER ';' {
    {
        // FILE << Graph
        struct Node* tn1 = astNewLeaf(IDENTIFIER, $1.s, $1.1);
        struct Node* tn3 = astNewLeaf(IDENTIFIER, $3.s, $3.1);
        sTableLookupId(tn1);
        sTableLookupId(tn3);
        $$ = astNewNode(AST_READ_GRAPH, 2, astAllChildren(2, tn1, tn3), $2.1);
    }
}
;

io_ext_list
: io_ext { $$ = $1; }
| io_ext_list io_ext { $$ = astNewNode(AST_PRINT_COMMA, 2, astAllChildren(2, $1, $2), $1->line); }

io_ext
: LIN assignment_expression {
    $$ = astNewNode(AST_PRINT, 1, astAllChildren(1, $2), $1.1);
}
;

/*****
 * EXPRESSIONS
 *****/

expression
: assignment_expression { $$ = $1; }
| expression ',' assignment_expression {
    $$ = astNewNode ( AST_COMMA, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

assignment_expression
: logical_OR_expression { $$ = $1; }
| postfix_expression assignment_operator assignment_expression {
    $$ = astNewNode ( $2.i, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

assignment_operator
: '=' { $$ .i = AST_ASSIGN; $$ .l = $1.1; }
| ADD_ASSIGN { $$ .i = ADD_ASSIGN; $$ .l = $1.1; }
| SUB_ASSIGN { $$ .i = SUB_ASSIGN; $$ .l = $1.1; }
| MUL_ASSIGN { $$ .i = MUL_ASSIGN; $$ .l = $1.1; }

```

```

    || DIV_ASSIGN      { $$i = DIV_ASSIGN; $$l = $1.1; }
    | APPEND           { $$i = APPEND;      $$l = $1.1; }
    ;

logical_OR_expression
: logical_AND_expression { $$ = $1; }
| logical_OR_expression OR logical_AND_expression {
    $$ = astNewNode ( OR, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

logical_AND_expression
: equality_expression { $$ = $1; }
| logical_AND_expression AND equality_expression {
    $$ = astNewNode ( AND, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

equality_expression
: relational_expression { $$ = $1; }
| equality_expression EQ relational_expression {
    $$ = astNewNode ( EQ, 2, astAllChildren(2, $1, $3), $2.1 );
}
| equality_expression NE relational_expression {
    $$ = astNewNode ( NE, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

relational_expression
: additive_expression { $$ = $1; }
| relational_expression LT additive_expression {
    $$ = astNewNode ( LT, 2, astAllChildren(2, $1, $3), $2.1 );
}
| relational_expression GT additive_expression {
    $$ = astNewNode ( GT, 2, astAllChildren(2, $1, $3), $2.1 );
}
| relational_expression LE additive_expression {
    $$ = astNewNode ( LE, 2, astAllChildren(2, $1, $3), $2.1 );
}
| relational_expression GE additive_expression {
    $$ = astNewNode ( GE, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

additive_expression
: multiplicative_expression { $$ = $1; }
| additive_expression ADD multiplicative_expression {
    $$ = astNewNode ( ADD, 2, astAllChildren(2, $1, $3), $2.1 );
}
| additive_expression SUB multiplicative_expression {
    $$ = astNewNode ( SUB, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

multiplicative_expression
: cast_expression { $$ = $1; }
| multiplicative_expression MUL cast_expression {
    $$ = astNewNode ( MUL, 2, astAllChildren(2, $1, $3), $2.1 );
}
| multiplicative_expression DIV cast_expression {
    $$ = astNewNode ( DIV, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

cast_expression
: unary_expression { $$ = $1; }
| '(' declaration_specifiers ')' cast_expression {
    $$ = astNewNode ( AST_CAST, 2, astAllChildren(2, $2, $4), $2->line );
}
;

unary_expression
: postfix_expression { $$ = $1; }
| unary_operator cast_expression {
    $$ = astNewNode ( $1.i, 1, astAllChildren(1, $2), $1.1 );
}
;

unary_operator
: ADD { $$i = AST_UNARY_PLUS; $$l = $1.1; }
| SUB { $$i = AST_UNARY_MINUS; $$l = $1.1; }

```



```

| '!' { $$i = AST_UNARY_NOT; $$l = $1l; }
;

postfix_expression
: primary_expression { $$ = $1; }
| primary_expression ':' primary_expression ARROW primary_expression {
    $$ = astNewNode ( ARROW, 3, astAllChildren(3, $1, $3, $5), $2.1 );
}
| IDENTIFIER '(' argument_expression_list ')' {
    struct Node* tn = astNewLeaf(IDENTIFIER, $1.s, $1.l);
    $$ = astNewNode(AST_FUNC_CALL, 2, astAllChildren(2, tn, $3), tn->line);
}
| IDENTIFIER '(' ')' {
    struct Node* tn = astNewLeaf(IDENTIFIER, $1.s, $1.l);
    $$ = astNewNode(AST_FUNC_CALL, 1, astAllChildren(1, tn), tn->line);
}
| postfix_expression PIPE pipe_property {
    $$ = astNewNode ( PIPE, 2, astAllChildren(2, $1, $3), $2.1 );
}
| postfix_expression '?' '[' no_type_check_on_dynamic_left dynamic_scope_left scope_in logical_OR_expression
    scope_out dynamic_scope_right no_type_check_on_dynamic_right ']' {
    $$ = astNewNode ( AST_MATCH, 3, astAllChildren(3, $1, $7, $8), $2.1 );
}
| postfix_expression '[' expression ']' {
    $$ = astNewNode ( AST_LIST_MEMBER, 2, astAllChildren(2, $1, $3), $2.1 );
}
| IDENTIFIER '.' IDENTIFIER {
    struct Node * tn1 = astNewLeaf(IDENTIFIER, $1.s, $1.l);
    struct Node * tn3 = astNewLeaf(IDENTIFIER, $3.s, $3.l);
    sTableLookupId(tn1);
    $$ = astNewNode ( AST_ATTRIBUTE, 2, astAllChildren(2, tn1, tn3), $2.1 );
    char * ctmp = tn3->lexval.sval;
    $$->child[1]->lexval.sval = strCatAlloc("", 2, ":", ctmp );
    free(ctmp);
}
| IDENTIFIER '.' graph_property {
    struct Node * tn1 = astNewLeaf(IDENTIFIER, $1.s, $1.l);
    sTableLookupId(tn1);
    $$ = astNewNode ( AST_GRAPH_PROP, 2, astAllChildren(2, tn1, $3), $2.1 );
}
;

primary_expression
: attribute {
    $$ = $1;
    if(isNoTypeCheck==0){ // Func_Literal // not used, JZ
        sTableLookupId($$); // Lookup ATTRIBUTE
    }
    else { // Match operator
        // As here we may use 'attribute' directly without declaration,
        // so it must be inserted into symbol table when first meets an 'attribute'
        $$->type = DYNAMIC_T; // 1. set type
        $$->symbol = sTableTryLookupId($$); // 2. try look up myself in symtable
        if ( $$->symbol==NULL ) { // if not exsist, insert it
            sTableInsertId($$, DYNAMIC_T);
        }
    }
}
| IDENTIFIER {
    $$ = astNewLeaf(IDENTIFIER, $1.s, $1.l);
    sTableLookupId($$); // Lookup IDENTIFIER in Symbol Table
}
| constant { $$ = $1; }
| STRING_LITERAL { $$ = astNewLeaf(STRING_LITERAL, $1.s, $1.l); }
| '(' expression ')' { $$ = $2; }
;

graph_property
: ALL_VERTICES { $$ = astNewLeaf(ALL_VERTICES, NULL, $1.l); }
| ALL_EDGES { $$ = astNewLeaf(ALL_EDGES, NULL, $1.l); }
;

pipe_property
: OUTCOMING_EDGES { $$ = astNewLeaf(OUTCOMING_EDGES, NULL, $1.l); }
| INCOMING_EDGES { $$ = astNewLeaf(INCOMING_EDGES, NULL, $1.l); }
| STARTING_VERTICES { $$ = astNewLeaf(STARTING_VERTICES, NULL, $1.l); }
| ENDING_VERTICES { $$ = astNewLeaf(ENDING_VERTICES, NULL, $1.l); }
;

argument_expression_list
: argument_expression { $$ = $1; }

```

```

    | argument_expression_list ',' argument_expression {
        $$ = astNewNode ( AST_COMMA, 2, astAllChildren(2, $1, $3), $2.1 );
    }
;

argument_expression
: assignment_expression {
    $$ = astNewNode ( AST_ARG_EXPS, 1, astAllChildren(1, $1), $1->line );
}
;

attribute
: AT IDENTIFIER{
    if (isDynamicScope==0) {
        ERRNO = ErrorDynamicAttributeUsedInNonDynamicScope;
        errorInfo(ERRNO, $2.1, "dynamic attribute '%s' is used in non-dynamic scope\n", $2.s);
    }
    $$ = astNewLeaf ( DYN_ATTRIBUTE, $2.s, $2.1 );
}
;

constant
: INTEGER_CONSTANT      { $$ = astNewLeaf(INTEGER_CONSTANT, $1.s, $1.1); }
| FLOAT_CONSTANT       { $$ = astNewLeaf(FLOAT_CONSTANT, $1.s, $1.1); }
| BOOL_TRUE            { $$ = astNewLeaf(BOOL_TRUE, NULL, $1.1); }
| BOOL_FALSE           { $$ = astNewLeaf(BOOL_FALSE, NULL, $1.1); }
| LENGTH '(' IDENTIFIER ')' {
    struct Node * tnode = astNewLeaf(IDENTIFIER, $3.s, $3.1);
    sTableLookupId(tnode);
    $$ = astNewNode(AST_LENGTH, 1, astAllChildren(1, tnode), $1.1 );
}
;

/*****
 *  DECLARATION
 *****/

function_literal_declaration
: function_literal_type_specifier func_declarator ':' declaration_specifiers '=' no_type_check_on_dynamic_left
  dynamic_scope_left compound_statement_no_scope dynamic_scope_right no_type_check_on_dynamic_right scope_out ';'
{
    $$ = astNewNode($1.i, 4, astAllChildren(4, $2, $4, $8,$11), $1.1);
    $$->typeCon = $2->typeCon;
    $$->scope[0] = $2->scope[0];
    $$->scope[1] = $2->scope[1];
    sTableDeclare($$);
}
| function_literal_type_specifier func_declarator ':' declaration_specifiers '=' no_type_check_on_dynamic_left
  dynamic_scope_left compound_statement_no_scope dynamic_scope_right no_type_check_on_dynamic_right scope_out
  error {
    astFreeTree($2);
    astFreeTree($4);
    astFreeTree($8);
    astFreeTree($11);
    $$ = NULL;
}
;

function_definition
: function_head compound_statement_no_scope scope_out{
    $$ = $1;
    $$->child[2] = $2;    // fill up the third field
    $$->child[3] = $3;
}
;

function_head
: declaration_specifiers func_declarator {
    $$ = astNewNode(AST_FUNC, 4, astAllChildren(4, $1, $2, NULL, NULL), $2->line); // third field empty
    $$->typeCon = $2->typeCon;
    $$->scope[0] = $2->scope[0];    // Scope Level
    $$->scope[1] = $2->scope[1];    // Scope Id
    sTableDeclare($$);    // We must declare before coming into compound stat, for recursive call
    // tmp no longer needed after here
}
;

function_literal_type_specifier
: FUNC_LITERAL      { $$i = FUNC_LITERAL; $$1 = $1.1; }
;

```

```

basic_type_specifier
: VOID      { int ttype = VOID_T;  $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| BOOLEAN   { int ttype = BOOL_T;   $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| INTEGER   { int ttype = INT_T;    $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| FLOAT     { int ttype = FLOAT_T;  $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| STRING    { int ttype = STRING_T; $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| VLIST     { int ttype = VLIST_T;  $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| ELIST     { int ttype = ELIST_T;   $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| VERTEX    { int ttype = VERTEX_T;  $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| EDGE      { int ttype = EDGE_T;   $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
| GRAPH     { int ttype = GRAPH_T;  $$= astNewLeaf(AST_TYPE_SPECIFIER, &(ttype), $1.1); }
;

declaration
: declaration_specifiers init_declarator_list ';' {
    $$ = astNewNode( AST_DECLARATION, 2, astAllChildren(2, $1, $2), $1->line );
    sTableDeclare($$);
}
| declaration_specifiers init_declarator_list error {
    astFreeTree($1);
    astFreeTree($2);
    $$ = NULL;
}
;

declaration_specifiers
: basic_type_specifier {
    $$ = $1;
}
;

init_declarator_list
: init_declarator {
    $$ = $1;
}
| init_declarator_list ',' init_declarator {
    $$ = astNewNode( AST_COMMA, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

init_declarator
: simple_declarator {
    $$ = $1;
}
| simple_declarator '=' initializer {
    $$ = astNewNode( AST_ASSIGN, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

simple_declarator
: IDENTIFIER {
    $$ = astNewLeaf(IDENTIFIER, $1.s, $1.1);
}
;

func_declarator
: func_id scope_in '(' parameter_list ')' {
    $$ = astNewNode( AST_FUNC_DECLARATOR, 2, astAllChildren(2, $1, $4), $1->line );
    // generate type Constructor for parameters
    $$->typeCon = astTypeConParaList( $4, NULL );
    $$->scope[0] = $1->scope[0]; // Scope Level
    $$->scope[1] = $1->scope[1]; // Scope Id
}
| func_id scope_in '(' ')' {
    $$ = astNewNode( AST_FUNC_DECLARATOR, 1, astAllChildren(1, $1 ), $1->line );
    $$->typeCon = astTypeConParaList( NULL, NULL );
    $$->scope[0] = $1->scope[0]; // Scope Level
    $$->scope[1] = $1->scope[1]; // Scope Id
}
;

func_id
: IDENTIFIER {
    $$ = astNewLeaf(IDENTIFIER, $1.s, $1.1);
}
;

parameter_list
: parameter_declaration { $$ = $1; }
| parameter_list ',' parameter_declaration {
    $$ = astNewNode( AST_COMMA, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

```

```

    }
    ;

parameter_declaration
: declaration_specifiers IDENTIFIER {
    struct Node* tn = astNewLeaf(IDENTIFIER, $2.s, $2.1);
    $$ = astNewNode( AST_PARA_DECLARATION, 2, astAllChildren(2, $1, tn), $1->line);
    sTableDeclare($$);
    $$->type = tn->type;
}
| declaration_specifiers attribute {
    // $$ = astNewNode( AST_PARA_DECLARATION, 2, astAllChildren(2, $1, $2), $1->line);
    // sTableDeclare($$);
    // $$->type = $2->type;
}
| function_literal_type_specifier IDENTIFIER {
    $$ = astNewNode( FUNC_LITERAL, 1, astAllChildren(1, astNewLeaf(IDENTIFIER, $2.s, $2.1)), $1.1);
    $$->type = FUNC_LITERAL_T;
}
;

initializer
: assignment_expression { $$ = $1; }
| '[' initializer_list '[' {
    $$ = astNewNode( AST_LIST_INIT, 1, astAllChildren(1, $2), $1.1 );
}
| '[' ']' {
    $$ = astNewNode( AST_LIST_INIT, 0, NULL, $1.1 );
}
;

initializer_list
: initializer { $$ = $1; }
| initializer_list ',' initializer {
    $$ = astNewNode( AST_COMMA, 2, astAllChildren(2, $1, $3), $2.1 );
}
;

/*****
* auxiliary nonterminal *
*****/

scope_in
: { sStackPush( sNewScopeId() );
    maxLevel = (maxLevel < sStackLevel) ? sStackLevel : maxLevel;
}
;

scope_out
: { $$ = astNewNode( AST_SCOPE_OUT, 0, NULL, -1);
    sStackPop(); }
;

dynamic_scope_left
: { isDynamicScope = 1; }
;

dynamic_scope_right
: { isDynamicScope = 0; }
;

no_type_check_on_dynamic_left
: { isNoTypeCheck = 1; }
;

no_type_check_on_dynamic_right
: { isNoTypeCheck = 0; }
;

%%

void yyerror(const char *s) {
    errorInfo(ErrorSyntax, yyval.LString.1, "%s\n", s);
}

void main_init(char * fileName) {
    init_util();
    sTableInit();
    tmpTableInit();
    sStackInit();
    isDynamicScope = 0;
    isNoTypeCheck = 0;
}

```

```

    maxLevel = 0;
    inLoop = 0;
    inFunc = -1;
    inFuncLiteral = -1;
    isFunc = 0;
    inMATCH = 0;
    existMATCH = 0;
    matchStaticVab = NULL;
    frontDeclExp = NULL;
    frontDeclExpTmp1 = NULL;
    existPIPE = 0;
    returnList = NULL;
    returnList2 = NULL;
    noReturn = NULL;
    noReturn2 = NULL;
    FuncParaList = NULL;
    OUTFILE = strCatAlloc("", 2, fileName, ".c");
}

void main_clean() {
    sTableDestroy();
    tmpTableDestroy();
    sStackDestroy();
    free(OUTFILE);
}

int main(int argc, char * const * argv) {
    if (argc<=1) { // missing file
        fprintf(stdout, "missing input file\n");
        exit(1);
    }
    main_init(argv[1]);
    yyin = fopen(argv[1], "r");

    yyparse();
    fclose(yyin);
    main_clean();
    if (ERRNO!=0) {
        fprintf(stderr, "error code = %d\n", ERRNO);
    }
    return ERRNO;
}

```

../src/Parser.y

```

// author : Jing

#ifndef ASTREE_H_NSBL_
#define ASTREE_H_NSBL_
/*****
 * Abstract Syntax Tree
 *****/
#include <stdio.h>
#include <glib/garray.h>
#include "Parser.tab.h" // definition of tokens
#include "SymbolTable.h"

#ifdef _DEBUG
#ifdef _AST_DEBUG_ALL
#define _AST_DEBUG_BASE
#define _AST_DEBUG_EXTRA
#define _AST_DEBUG_MEMORY
#endif
#ifdef _AST_DEBUG
#define _AST_DEBUG_BASE
#endif
#endif

typedef union {
    bool      bval;
    int       ival;
    float     fval;
    char*     sval;
} val_t;

struct Node {
    int      token; // can also be operator
    int      type;  // see SymbolTable.h
    GArray*  typeCon; // type constructor
    int      nch;    // number of children
    struct Node** child; // if Leaf, NULL

```

```

    val_t          lexval;          // store lexeme value
    SymbolTableEntry* symbol;        // default NULL
    long long      line;            // # line in source
    char*          code;            // target code
    char*          codetmp;         // target code in c's global
    ScopeId        scope[2];        // scopeLevel, scopeId
    int            tmp[10];          // temp storage
};

/** create a Leaf of AST, ptr is the pointer to the lexemeval */
struct Node* ast_new_leaf(int token, void * ptr, long long l);

/** create a node of AST */
struct Node* ast_new_node(int token, int nch, struct Node** child, long long l);

/** pack all children in order */
struct Node** ast_all_children(int n, ...);

/** get the lexval.field */
#define ast_leaf_val(leaf,field) leaf.lexval.field

/** free subtree */
void ast_free_tree(struct Node* node);

/** find the leftmost child */
struct Node* ast_leftmost_child(struct Node* node);

/** output functions */
void ast_output_node(struct Node* node, FILE* out, const char * sep);
void ast_output_tree(struct Node* node, FILE* out, int level);

/** type construct */
GArray* ast_type_construct_parameter_list(struct Node* node, GArray* ga);
GArray* ast_type_construct_argument_expression_list(struct Node* node, GArray* ga);
void ast_free_type_construct(GArray* ga);

/*****
 * Call Wrapper *
 *****/

#define astNewLeaf(t,p,l)      ast_new_leaf( t,(void*) p,l )
#define astNewNode(t,n,c,l)   ast_new_node( t,n,c,l )
#define astAllChildren        ast_all_children
#define astFreeTree           ast_free_tree
#define astLeftmostNode       ast_leftmost_child
#define astOutNode             ast_output_node
#define astOutTree             ast_output_tree

#define astTypeConParaList     ast_type_construct_parameter_list
#define astTypeConArgList      ast_type_construct_argument_expression_list
#define astFreeTypeCon         ast_free_type_construct

#endif

```

../src/ASTree.h

```

/*****
- Original Built-up : Jing
- Later Append: Chantal, Kunal, Lixing
 *****/

#include "ASTree.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include "util.h"
#include "CodeGenUtil.h"
#include "global.h"

#ifdef _AST_DEBUG_BASE
extern FILE* DEBUGIO;
#endif

/** create a leaf in AST */
struct Node* ast_new_leaf(int token, void * ptr, long long line) {
    struct Node* node = (struct Node *) malloc ( sizeof (struct Node) ); // free in ast_free_tree
    node->typeCon = NULL;
    node->nch = 0; // Leaf has no child
    node->child = NULL;
    node->symbol = NULL; // default null
}

```

```

node->line = line;           // line # in source
node->code = NULL;           // no code assigned
node->codetmp = NULL;
node->scope[0] = sStackLevel; // scope Level
node->scope[1] = sStackTopId; // scope ID
node->tmp[0] = 0;
switch (token) {
case INTEGER_CONSTANT :
    node->token = INTEGER_CONSTANT;
    node->type = INT_T;
    node->lexval.ival = atoi( (const char *) ptr );
    node->code = strCatAlloc(" ", 1, (const char *)ptr);
    break;
case FLOAT_CONSTANT :
    node->token = FLOAT_CONSTANT;
    node->type = FLOAT_T;
    node->lexval.fval = atof( (const char *) ptr );
    node->code = strCatAlloc(" ", 1, (const char *)ptr);
    break;
case BOOL_TRUE :
    node->token = BOOL_TRUE;
    node->type = BOOL_T;
    node->lexval.bval = true;
    node->code = strCatAlloc(" ", 1, "true");
    break;
case BOOL_FALSE :
    node->token = BOOL_FALSE;
    node->type = BOOL_T;
    node->lexval.bval = false;
    node->code = strCatAlloc(" ", 1, "false");
    break;
case STRING_LITERAL :
    node->token = STRING_LITERAL;
    node->type = STRING_T;
    node->lexval.sval = (char *) ptr;
    node->code = strCatAlloc(" ", 3, "g_string_new ( ", (char *)ptr, " )");
    break;
case IDENTIFIER :
    node->token = IDENTIFIER;
    node->type = UNKNOWN_T;
    node->lexval.sval = (char *) ptr;
    break;
case DYN_ATTRIBUTE :
    node->token = DYN_ATTRIBUTE;
    node->type = UNKNOWN_T;
    node->lexval.sval = (char *) ptr;
    break;
case AST_TYPE_SPECIFIER :
    node->token = AST_TYPE_SPECIFIER;
    node->type = UNKNOWN_T;
    node->lexval.ival = *((int *) ptr);
    break;
case ALL_VERTICES :
    node->token = ALL_VERTICES; break;
case ALL_EDGES :
    node->token = ALL_EDGES; break;
case OUTCOMING_EDGES :
    node->token = OUTCOMING_EDGES; break;
case INCOMING_EDGES :
    node->token = INCOMING_EDGES; break;
case STARTING_VERTICES :
    node->token = STARTING_VERTICES; break;
case ENDING_VERTICES :
    node->token = ENDING_VERTICES; break;
default:
    fprintf(stderr, "ast_new_leaf: unknown token: %d\n", token);
}
#ifdef _AST_DEBUG_BASE
    debugInfo("ast_new_leaf :: create ");
    ast_output_node(node, DEBUGIO, "\n");
#endif
return node;
}

struct Node** ast_all_children(int n, ...){
    if (n<=0) return NULL;
    int i;
    va_list args;
    va_start (args, n);
    struct Node** child = (struct Node**) malloc ( sizeof(struct Node *) * n ); // free in ast_free_tree
    for(i=0; i<n; ++i) { child[i] = va_arg(args, struct Node *); }
}

```

```

    va_end(args);
    return child;
}

struct Node* ast_new_node(int token, int nch, struct Node** child, long long line){
    struct Node* node = (struct Node *) malloc ( sizeof (struct Node) ); // free in ast_free_tree

    node->token = token;
    node->type = UNKNOWN_T; // default
    node->typeCon = NULL;
    node->nch = nch; // assign children
    node->child = child;
    node->symbol = NULL; // default NULL
    node->line = line; // line # in source (for corresponding token)
    node->code = NULL;
    node->codetmp = NULL;
    node->scope[0] = sStackLevel;
    node->scope[1] = sStackTopId;
    node->tmp[0] = 0;
#ifdef _AST_DEBUG_EXTRA
    debugInfo("ast_new_node :: create \n");
    debugInfo("==DEBUG INFO==\n");
    ast_output_tree(node, stdout, 0);
#endif
    return node;
}

void ast_free_tree(struct Node* node) {
    if ( node == NULL ) return;
#ifdef _AST_DEBUG_MEMORY
    debugInfo("TO FREE node:");
    ast_output_node(node, DEBUGIO, "\n");
#endif
    /* free sval */
    if ( node->token == STRING_LITERAL || node->token == IDENTIFIER ) {
#ifdef _AST_DEBUG_MEMORY
        debugInfo("FREE sval: %s\n", node->lexval.sval);
#endif
        free(node->lexval.sval); // malloc by LexAly.1
        node->lexval.sval = NULL;
    }
    /* free code */
    if ( node->code != NULL ) {
#ifdef _AST_DEBUG_MEMORY
        debugInfo("FREE code: %s\n", node->code);
#endif
        free(node->code); node->code = NULL;
    }
    if ( node->codetmp != NULL ) {
#ifdef _AST_DEBUG_MEMORY
        debugInfo("FREE codetmp: %s\n", node->codetmp);
#endif
        free(node->codetmp); node->codetmp = NULL;
    }
    /* if child exists, free child first */
    if ( node->nch > 0 && node->child != NULL ) {
        // free children
        int i; for(i=0; i<node->nch; ++i) {
            ast_free_tree( node->child[i] );
            node->child[i] = NULL;
        }
        // free child ptr array
#ifdef _AST_DEBUG_MEMORY
        debugInfo("FREE child ptrs in ");
        ast_output_node(node, DEBUGIO, "\n");
#endif
        free(node->child); node->child = NULL;
    }
    else if (node->nch > 0 || node->child != NULL) {
        fprintf(stderr, "ERROR:: ast_free_tree :: nch does NOT match child! code bug detected!!\n ");
    }
    /* free myself */
#ifdef _AST_DEBUG_MEMORY
    debugInfo("FREE this node : %d", node->token);
    ast_output_node(node, DEBUGIO, "\n");
    fflush(DEBUGIO);
#endif
    free(node); node = NULL;
    return;
}

```



```

void ast_output_node(struct Node* node, FILE* out, const char * sep) {
    if(node==NULL) return;
    fprintf(out, "%lld::", node->line);
    switch (node->token) {
        case INTEGER_CONSTANT :
            fprintf(out, "Node<INT>      : lexval = %d", node->lexval.ival);break;
        case FLOAT_CONSTANT :
            fprintf(out, "Node<FLOAT>   : lexval = %f", node->lexval.fval);break;
        case BOOL_TRUE :
            fprintf(out, "Node<TRUE>  "); break;
        case BOOL_FALSE :
            fprintf(out, "Node<FALSE> "); break;
        case STRING_LITERAL :
            fprintf(out, "Node<STRING> : lexval = %s", node->lexval.sval);break;
        case IDENTIFIER :
            fprintf(out, "Node<ID>      : lexval = %s type = %d ", node->lexval.sval, node->type);
            if(node->symbol!=NULL) fprintf(out, "bind = %s", node->symbol->bind);
            break;
        case DYN_ATTRIBUTE:
            fprintf(out, "node<DYN_ATTR>: lexval = %s type = %d ", node->lexval.sval, node->type);
            if(node->symbol!=NULL) fprintf(out, "bind = %s", node->symbol->bind);
            break;
        case AST_TYPE_SPECIFIER :
            fprintf(out, "Node<TYPE>      : lexval = %s", s_table_type_name(node->lexval.ival));break;
        case AST_DECLARATION :
            fprintf(out, "Node<DECLAR>");break;
        case AST_FUNC_DECLARATOR :
            fprintf(out, "Node<FUNC_DECLARATOR>");break;
        case BELONG :
            fprintf(out, "Node<BELONG>");break;
        case AST_PARA_DECLARATION :
            fprintf(out, "Node<PARA_DECLARATION>");break;
        case AST_LIST_INIT :
            fprintf(out, "Node<LIST_INIT>");break;
        case AST_COMMA :
            fprintf(out, "Node<COMMA>");break;
        case AST_ASSIGN :
            fprintf(out, "Node<ASSIGN>");break;
        case ADD_ASSIGN :
            fprintf(out, "Node<ADD_ASSIGN>");break;
        case SUB_ASSIGN :
            fprintf(out, "Node<SUB_ASSIGN>");break;
        case MUL_ASSIGN :
            fprintf(out, "Node<MUL_ASSIGN>");break;
        case DIV_ASSIGN :
            fprintf(out, "Node<DIV_ASSIGN>");break;
        case APPEND :
            fprintf(out, "Node<APPEND>");break;
        case OR :
            fprintf(out, "Node<OR>");break;
        case AND :
            fprintf(out, "Node<AND>");break;
        case EQ :
            fprintf(out, "Node<EQ>");break;
        case NE :
            fprintf(out, "Node<Ne>");break;
        case LT :
            fprintf(out, "Node<LT>");break;
        case GT :
            fprintf(out, "Node<GT>");break;
        case LE :
            fprintf(out, "Node<LE>");break;
        case GE :
            fprintf(out, "Node<GE>");break;
        case ADD :
            fprintf(out, "Node<ADD>");break;
        case SUB :
            fprintf(out, "Node<SUB>");break;
        case MUL :
            fprintf(out, "Node<MUL>");break;
        case DIV :
            fprintf(out, "Node<DIV>");break;
        case AST_CAST :
            fprintf(out, "Node<CAST>");break;
        case AST_UNARY_PLUS :
            fprintf(out, "Node<UNARY_PLUS>");break;
        case AST_UNARY_MINUS :
            fprintf(out, "Node<UNARY_MINUS>");break;
        case AST_UNARY_NOT :
            fprintf(out, "Node<UNARY_NOT>");break;
        case ARROW :

```

```

        fprintf(out, "Node<ARROW>");break;
case PIPE :
    fprintf(out, "Node<PIPE>");break;
case AST_MATCH :
    fprintf(out, "Node<MATCH>");break;
case AST_ATTRIBUTE :
    fprintf(out, "Node<ATTRIBUTE>");break;
case AST_GRAPH_PROP :
    fprintf(out, "Node<GRAPH_PROP>");break;
case AST_STAT_LIST :
    fprintf(out, "Node<STAT_LIST>");break;
case AST_COMP_STAT :
    fprintf(out, "Node<COMP_STAT>");break;
case AST_COMP_STAT_NO_SCOPE :
    fprintf(out, "Node<COMP_STAT_NO_SCOPE>");break;
case AST_EXT_STAT_COMMA :
    fprintf(out, "Node<EXT_STAT_COMMA>");break;
case AST_FUNC :
    fprintf(out, "Node<FUNCTION>");break;
case FUNC_LITERAL:
    fprintf(out, "Node<FUNC_LITERAL>");break;
case AST_IF_STAT :
    fprintf(out, "Node<IF_STAT>");break;
case AST_IFELSE_STAT :
    fprintf(out, "Node<IFELSE_STAT>");break;
case AST_WHILE :
    fprintf(out, "Node<WHILE_STAT>");break;
case AST_FOREACH :
    fprintf(out, "Node<FOREACH_STAT>");break;
case AST_FOR :
    fprintf(out, "Node<FOR_STAT>");break;
case AST_JUMP_CONTINUE:
    fprintf(out, "Node<CONTINUE>");break;
case AST_JUMP_BREAK:
    fprintf(out, "Node<BREAK>");break;
case AST_JUMP_RETURN:
    fprintf(out, "Node<RETRUN>");break;
case AST_FUNC_CALL:
    fprintf(out, "Node<FUNC_CALL>");
    if (node->symbol!=NULL) fprintf(out, "bind = %s", node->symbol->bind);
    break;
case ALL_VERTICES :
    fprintf(out, "Node<ALL_VERTICES>");break;
case OUTCOMING_EDGES :
    fprintf(out, "Node<OUTEDGES>");break;
case STARTING_VERTICES :
    fprintf(out, "Node<STARTING_VERTICES>");break;
case ENDING_VERTICES :
    fprintf(out, "Node<ENDING_VERTICES>");break;
case ALL_EDGES :
    fprintf(out, "Node<ALL_EDGES>");break;
case INCOMING_EDGES:
    fprintf(out, "Node<INCOMING_EDGES>");break;
case AT :
    fprintf(out, "Node<AT_ATTRIBUTE>");break;
case AST_ARG_EXPS :
    fprintf(out, "Node<ARGUMENT_EXP>");break;
case AST_EXP_STAT :
    fprintf(out, "Node<EXP_STAT>");break;
case AST_ERROR :
    fprintf(out, "Node<ERROR>");break;
case AST_PRINT :
    fprintf(out, "Node<AST_PRINT>");break;
case AST_PRINT_COMMA :
    fprintf(out, "Node<AST_PRINT_COMMA>");break;
case AST_PRINT_STAT :
    fprintf(out, "Node<AST_PRINT_STAT>");break;
case AST_READ_GRAPH :
    fprintf(out, "Node<AST_READ_GRAPH>");break;
case AST_WRITE_GRAPH :
    fprintf(out, "Node<AST_WRITE_GRAPH>");break;
case AST_LIST_MEMBER :
    fprintf(out, "Node<AST_LIST_MEMBER>");break;
case AST_LENGTH :
    fprintf(out, "Node<AST_LENGTH>");break;
case AST_SCOPE_OUT :
    fprintf(out, "Node<AST_SCOPE_OUT>");break;
default :
    fprintf(out, "Node<UNKNOWN> !!!!!!!!!!!!!!!");
}
fprintf(out, " lvl=%d ", node->scope[0]);

```

```

    if(node->code != NULL) fprintf(out, " \n code = '%s'", node->code);
    if(node->codetmp != NULL) fprintf(out, "\n codetmp = '%s'", node->codetmp);
    fprintf(out, "%s", sep);
    return;
}

/** preorder output */
void ast_output_tree(struct Node* node, FILE* out, int level) {
    int i;
    int indent = level;
    if(node == NULL) return;
    while(indent-->0){
        fprintf(out, " ");
    }
    fprintf(out, "Tree<%d>:", level);
    ast_output_node(node, out, "\n");
    for(i=0; i<node->nch; ++i) {
        ast_output_tree(node->child[i], out, level+1);
    }
}

/** find the leftmost child */
struct Node* ast_leftmost_child(struct Node* node) {
    if( node == NULL ) return NULL;
    if( node->nch <= 0 ) return node;
    return ast_leftmost_child(node->child[0]);
}

/** create type construct for parameter_list */
GArray* ast_type_construct_parameter_list(struct Node* node, GArray* ga) {
    if(ga==NULL) ga = g_array_new (1,1,sizeof(int)); // destroy in SymbolTable.c
    if(node==NULL) return ga;
    if(node->token == AST_COMMA) {
        ast_type_construct_parameter_list(node->child[0],ga); // left
        ast_type_construct_parameter_list(node->child[1],ga); // right
    }
    else if(node->token == AST_PARA_DECLARATION ||
            node->token == FUNC_LITERAL ) {
        g_array_append_vals ( ga, (gconstpointer) & (node->type), 1 );
    }
    else {
        fprintf(stderr, "Error: ast_type_construct_parameter_list :: see unknown token : %d\n", node->token);
    }
    return ga;
}

/** create type construct for argument_expression_list */
GArray* ast_type_construct_argument_expression_list(struct Node* node, GArray* ga) {
    if(ga==NULL) ga = g_array_new (1,1,sizeof(int)); // destroy by ast_free_type_construct
    if(node==NULL) return ga;
    if(node->token == AST_COMMA) {
        ast_type_construct_argument_expression_list(node->child[0],ga); // left
        ast_type_construct_argument_expression_list(node->child[1],ga); // right
    }
    else if(node->token == AST_ARG_EXPS) {
        g_array_append_vals ( ga, (gconstpointer) & (node->type), 1 );
    }
    else {
        fprintf(stderr, "Error: ast_type_construct_argument_expression_list :: see unknown token : %d\n", node->token);
    }
    return ga;
}

void ast_free_type_construct(GArray* ga) {
    if(ga!=NULL) {
        g_array_free(ga, 1);
        ga = NULL;
    }
}

```

../src/ASTree.c

3 Symbol Table

```

// author: Jing
#ifdef SYMBOLTABL_H_NSBL_
#define SYMBOLTABL_H_NSBL_

```

```

#include <glib/ghash.h>
#include <glib/garray.h>
#include <stdio.h>
#include <stddef.h>
#include <stdbool.h>
#include "Error.h"

/** LENGTH */
#define LEXEME_LENGTH      128
#define BIND_LENGTH        128
#define KEY_LENGTH         256
#define S_STACK_INIT_LENGTH 128

/** ALL TYPES */
#include "type.h"

typedef int ScopeId;
typedef GHashTable SymbolTable;
typedef char SymbolTableKey[KEY_LENGTH];
typedef char Lexeme[LEXEME_LENGTH];
typedef char Binding[BIND_LENGTH];

/** Entry of Symbol Table */
typedef struct {
    Lexeme      lex;
    int         type;           // basic var type, FUNC, FUNC_LITERAL
    int         rtype;         // return type
    GArray*     typeCon;       // var : NULL
                                // fun : ( returnType, paraType1, ...)
    ScopeId     scope[2];      // level, Id
    SymbolTableKey key;
    Binding     bind;
    long long   line;
} SymbolTableEntry;

typedef struct {
    GArray*     stack;
    int         top;           // pointing to top of stack, initial value is 0.
    int         present;       // a pointer, initial value is 0.
} SymbolTableStack;

/** Methods */
/*****
 * ATTENTION:
 * - gpointer is type of entry for GHashTable, i.e. void *
 *****/

void      s_table_init      (SymbolTable** s_table);
void      s_table_destroy   (SymbolTable* s_table);
int        s_table_insert   (SymbolTable* s_table, SymbolTableEntry* the_entry);
bool       s_table_remove   (SymbolTable* s_table, SymbolTableEntry* the_entry, bool keepEntry);
SymbolTableEntry* s_table_lookup (SymbolTable* s_table, SymbolTableKey key);
int        s_table_check_key_exsit (SymbolTable* table, SymbolTableKey key);
void       s_table_show     (SymbolTable* table, FILE* out);
void       s_table_max_level (SymbolTable* table, int* mlevel);

GList*     s_table_all_variables_in_scope (SymbolTable* table, ScopeId sid, int type);

char*      s_table_type_name (int type);
char*      s_table_short_type_name (int type);
ScopeId    s_table_new_scopeid ();

int         s_stack_init    (SymbolTableStack** s_stack);
int         s_stack_destroy  (SymbolTableStack* s_stack);
int         s_stack_push    (SymbolTableStack* s_stack, ScopeId scopeid);
ScopeId     s_stack_pop     (SymbolTableStack* s_stack);
ScopeId     s_stack_top_id  (SymbolTableStack* s_stack);
ScopeId     s_stack_down    (SymbolTableStack* s_stack);
int         s_stack_reset   (SymbolTableStack* s_stack);

SymbolTableEntry* s_new_var_entry (Lexeme lex, int type, long long line);
SymbolTableEntry* s_new_fun_entry (Lexeme lex, int type, int rtype, GArray* typeCon, ScopeId sLevel, ScopeId
    sId, long long line);
void          s_destroy_entry (gpointer dummy1, gpointer entry, gpointer dummy2);
void          s_show_entry    (gpointer key, gpointer entry, gpointer out);
void          s_show_typeCon  (GArray* tc, FILE* out);
int           s_new_key       (Lexeme lex, ScopeId scope2, SymbolTableKey key);
int           s_new_bind      (SymbolTableEntry* entry, Binding bind);
int           s_entry_copy    ( SymbolTableEntry * dest, SymbolTableEntry * source );

```

```

int                tmp_new_key      (Lexeme lex, ScopeId scope2, SymbolTableKey key);
int                tmp_new_bind     (SymbolTableEntry* entry, Binding bind);
SymbolTableEntry*  tmp_new_var_entry (Lexeme lex, int type, ScopeId sid);

/*****
 * Call Wrapper *
*****/

#define sTableInit()      s_table_init( &s_table )
#define sTableDestroy()  s_table_destroy( s_table )
#define sTableInsert(e)   s_table_insert( s_table, e )
#define sTableRemove(e)   s_table_remove( s_table, e, false )
#define sTableRemoveKeep(e) s_table_remove( s_table, e, true )
#define sTableLookup(k)   s_table_lookup( s_table, k )
#define sTableShow(o)     s_table_show( s_table, o )
#define sTableMaxLevel(l) s_table_max_level( s_table, l )
#define sTableAllVarScope(s,t) s_table_all_variables_in_scope( s_table, s, t )

#define sTypeName(t)      s_table_type_name(t)
#define sShortTypeName(t) s_table_short_type_name(t)
#define sNewScopeId()     s_table_new_scopeid()

#define sStackInit()      s_stack_init( &s_stack )
#define sStackDestroy()   s_stack_destroy( s_stack )
#define sStackPush(a)     s_stack_push( s_stack, a )
#define sStackPop()       s_stack_pop( s_stack )
#define sStackDown()      s_stack_down( s_stack )
#define sStackReset()     s_stack_reset( s_stack )
#define sStackLevel       s_stack->top
#define sStackTopId       s_stack_top_id( s_stack )

#define sNewVarEty(l,t,ll) s_new_var_entry(l,t,ll)
#define sNewFunEty(l,t,rt,tc,s1,sd,ll) s_new_fun_entry(l,t,rt,tc,s1,sd,ll)
#define sDestroyEntry(e)   s_destroy_entry(NULL,e,NULL)
#define sNewKey(l,s2,k)    s_new_key(l,s2,k)
#define sNewBind(e,b)      s_new_bind(e,b)

#define tmpTableInit()     s_table_init( &tmp_table )
#define tmpTableDestroy()  s_table_destroy( tmp_table )
#define tmpTableInsert(e)   s_table_insert( tmp_table, e )
#define tmpTableRemove(e)   s_table_remove( tmp_table, e, false )
#define tmpTableRemoveKeep(e) s_table_remove( tmp_table, e, true )
#define tmpTableLookup(k)   s_table_lookup( tmp_table, k )
#define tmpTableShow(o)     s_table_show( tmp_table, o )
#define tmpTableMaxLevel(l) s_table_max_level( tmp_table, l )
#define tmpTableAllVarScope(s,t) s_table_all_variables_in_scope( tmp_table, s, t )
#define tmpNewVarEty(l,t,s) tmp_new_var_entry(l,t,s)

#define sEntryCopy(d,s)    s_entry_copy(d,s)
#endif

```

../src/SymbolTable.h

```

// author : Jing
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SymbolTable.h"
#ifdef _DEBUG
extern FILE* DEBUGIO;
#endif

/** Global Variables */
SymbolTable* s_table;
SymbolTableStack* s_stack;
int isDynamicScope;
int isNoTypeCheck;
int maxLevel;
SymbolTable* tmp_table;

// init Symbol Table
void s_table_init (SymbolTable** table) {
    *table = g_hash_table_new (g_str_hash, g_str_equal);
    debugInfo("Symbol Table Initialized.\n");
}

// destroy Symbol Table
void s_table_destroy (SymbolTable* table) {
    debugInfo("Try to destroy Symbol Table\n");
    g_hash_table_foreach(table, &s_destroy_entry, NULL);
}

```

```

    g_hash_table_destroy ( table );
}

// insert an entry into ST
int s_table_insert (SymbolTable* table, SymbolTableEntry* entry) {
    if ( s_table_check_key_exsit(table, entry->key) )
        return ErrorSymbolTableKeyAlreadyExsit;
    g_hash_table_insert(table, (gpointer) (entry->key), (gpointer) entry);
    return 0;
}

// remove an entry (with key) from ST
bool s_table_remove (SymbolTable* table, SymbolTableEntry* entry, bool keepEntry) {
    bool flag = g_hash_table_remove(table, (gpointer) (entry->key));
    if(!keepEntry) sDestroyEntry ( entry );
    return flag;
}

// find an entry with key
SymbolTableEntry* s_table_lookup (SymbolTable* table, SymbolTableKey key) {
    return (SymbolTableEntry*) g_hash_table_lookup(table, (gpointer) key);
}

// check if key exist
int s_table_check_key_exsit (SymbolTable* table, SymbolTableKey key) {
    if ( g_hash_table_lookup(table, (gpointer) key) != NULL ) return 1;
    return 0;
}

// Output an entry
void s_show_entry (gpointer key, gpointer entry, gpointer out) {
    SymbolTableEntry * e = (SymbolTableEntry*) entry;
    fprintf( (FILE*) out, "%10s %3d %3d %3d %3d %15s %15s %4lld ||",
        e->lex, e->type, e->rtype, e->scope[0], e->scope[1], e->key, e->bind, e->line );
    s_show_typeCon(e->typeCon, out);
    fprintf( (FILE*) out, "\n" );
}

void s_show_typeCon (GArray* tc, FILE* out){
    if (tc==NULL) return;
    int ii, ll = tc->len;
    for (ii=0; ii<ll; ++ii)
        fprintf( (FILE*) out, " %3d", g_array_index(tc, ScopeId, ii) );
}

// show entire ST
void s_table_show (SymbolTable* table, FILE* out) {
    fprintf(out, "%10s %3s %3s %3s %15s %15s %4s || %15s\n",
        "Lexeme", "T", "RT", "L", "Sp", "Key", "Binding", "Line", "Func Parameters");
    g_hash_table_foreach(table, &s_show_entry, (gpointer) out);
}

void s_entry_compare_level (gpointer key, gpointer entry, gpointer mlevel){
    SymbolTableEntry* e = (SymbolTableEntry*) entry;
    if ( e->scope[0] > *(int *) mlevel ) *(int *) mlevel = e->scope[0];
}

void s_table_max_level (SymbolTable* table, int* mlevel) {
    g_hash_table_foreach(table, &s_entry_compare_level, (gpointer) mlevel);
}

GList* s_table_all_variables_in_scope (SymbolTable* table, ScopeId sid, int type) {
    GList* gl = NULL;
    GList* vals = g_hash_table_get_values( table );
    int i, l = g_list_length( vals );
    for ( i=0; i<l; ++i ) {
        SymbolTableEntry * e = (SymbolTableEntry *) g_list_nth_data( vals, i );
        if ( e->scope[1] == sid && e->type == type ) {
#ifdef _DEBUG
            debugInfo("s_table_all_variables_in_scope: sid = %d, type = %d, %s\n", sid, type, e->bind);
#endif
            gl = g_list_append( gl, (gpointer) e );
        }
    }
    g_list_free(vals);
    return gl;
}

// convert type MACRO to char *
char* s_table_type_name (int type) {
    switch (type) {
        case VOID_T: return "void";
    }
}

```

```

        case BOOL_T:           return "bool";
        case INT_T:            return "int";
        case FLOAT_T:          return "float";
        case STRING_T:         return "StringType";
        case VLIST_T:          return "ListType";
        case ELIST_T:           return "ListType";
        case VERTEX_T:          return "VertexType";
        case EDGE_T:           return "EdgeType";
        case GRAPH_T:           return "GraphType";
        case DYNAMIC_T:         return "dyn";
        case FUNC_T:            return "func";
        case FUNC_LITERAL_T:    return "fl";
        case DYN_BOOL_T:        return "D_bool";
        case DYN_INT_T:         return "D_int";
        case DYN_FLOAT_T:       return "D_float";
        case DYN_STRING_T:      return "D_string";
        case DYN_ATTR_T:        return "Attribute";
        case UNKNOWN_T:         return "UNKOWN";
        case NOT_AVAIL:         return "NOT_AVAL";
        default:                return NULL;
    }
}

char* s_table_short_type_name (int type) {
    switch (type) {
        case VOID_T:           return "v";
        case BOOL_T:           return "b";
        case INT_T:            return "i";
        case FLOAT_T:          return "f";
        case STRING_T:         return "s";
        case VLIST_T:          return "vl";
        case ELIST_T:           return "el";
        case VERTEX_T:          return "v";
        case EDGE_T:           return "e";
        case GRAPH_T:           return "g";
        case DYNAMIC_T:         return "DD";
        case FUNC_T:            return "fc";
        case FUNC_LITERAL_T:    return "fl";
        case DYN_BOOL_T:        return "Db";
        case DYN_INT_T:         return "Di";
        case DYN_FLOAT_T:       return "Df";
        case DYN_STRING_T:      return "Ds";
        default:                return NULL;
    }
}

// get new binder Id
int s_table_new_bindid () {
    static int tid = 0;
    return tid++;
}

// get new scope Id
ScopeId s_table_new_scopeid () {
    static ScopeId sid = 0;
    return sid++;
}

// init Scope Stack
int s_stack_init (SymbolTableStack** stack) {
    SymbolTableStack* tstack = (SymbolTableStack*) malloc ( sizeof(SymbolTableStack) );
    tstack->stack = g_array_new (1,1,sizeof(ScopeId));
    tstack->top = -1;
    tstack->present = -1;
    s_stack_push( tstack, s_table_new_scopeid () );
    *stack = tstack;
    return 0;
}

// destroy Scope Stack
int s_stack_destroy (SymbolTableStack* stack) {
    g_array_free ( stack->stack, 1 );
    free(stack);
    return 0;
}

// push in one Scope Id
int s_stack_push (SymbolTableStack* stack, ScopeId sid) {
    stack->stack = g_array_append_vals ( stack->stack, (gconstpointer) (&sid), 1 );
    stack->top ++;
}

```

```

    return 0;
}

// pop out
ScopeId s_stack_pop (SymbolTableStack* stack) {
    ScopeId val = g_array_index ( stack->stack, ScopeId, stack->top );
    stack->stack = g_array_remove_index ( stack->stack, stack->top );
    stack->top --;
    return val;
}

// get the Scope Id on the top of stack
ScopeId s_stack_top_id (SymbolTableStack* stack) {
    return g_array_index ( stack->stack, ScopeId, stack->top );
}

// return scope id pointed by 'present' and 'present' move downward
ScopeId s_stack_down (SymbolTableStack* stack) {
    if(stack->present == -1) return -1; //stack bottom
    return g_array_index ( stack->stack, ScopeId, (stack->present)-- );
}

// reset ptr 'present' to the top
int s_stack_reset (SymbolTableStack* stack) {
    stack->present = stack->top;
    return 0;
}

// create variable Symbol Table entry
SymbolTableEntry* s_new_var_entry (Lexeme lex, int type, long long line ) {
    if( type==FUNC_T || type==FUNC_LITERAL_T ) {
        fprintf(stderr, "Hey wrong call to s_new_var_entry!\n");
        return NULL;
    }
    SymbolTableEntry* entry = (SymbolTableEntry*) malloc ( sizeof (SymbolTableEntry) );
    strcpy ((char *) entry->lex, lex);
    entry->line = line;
    entry->type = type;
    entry->rtype = NOT_AVAIL;
    entry->typeCon = NULL;
    entry->scope[0] = sStackLevel;
    entry->scope[1] = sStackTopId;
    s_new_key ( entry->lex, entry->scope[1], entry->key );
    s_new_bind ( entry, entry->bind );
    return entry;
}

// create function Symbol Table entry
SymbolTableEntry* s_new_fun_entry (Lexeme lex, int type, int rtype, GArray* typeCon, ScopeId sLevel, ScopeId sId, long
    long line ) {
    if( type!=FUNC_T && type!=FUNC_LITERAL_T ) {
        fprintf(stderr, "Hey wrong call to s_new_fun_entry!\n");
        return NULL;
    }
    SymbolTableEntry* entry = (SymbolTableEntry*) malloc ( sizeof (SymbolTableEntry) );
    strcpy ((char *) entry->lex, lex);
    entry->line = line;
    entry->type = type;
    entry->rtype = rtype;
    entry->typeCon = typeCon;
    entry->scope[0] = sLevel;
    entry->scope[1] = sId;
    s_new_key ( entry->lex, entry->scope[1], entry->key );
    s_new_bind ( entry, entry->bind );
    return entry;
}

// destroy Symbol Table entry
void s_destroy_entry ( gpointer dummy1, gpointer entry, gpointer dummy2 ) {
#ifdef _DEBUG
    debugInfo("Destroy Entry: ");
    s_show_entry(NULL, entry, DEBUGIO);
#endif
    SymbolTableEntry* e = ( SymbolTableEntry* ) entry ;
    if (e->typeCon != NULL) {
        g_array_free( e->typeCon, 1 );
    }
#ifdef _DEBUG
    debugInfoExt(" >> destroy typeCon... \n");
#endif
}

```



```

    free( e );
}

// create new key
int s_new_key ( Lexeme lex, ScopeId scope, SymbolTableKey key) {
    sprintf( key, "%s%d\0", lex, scope );
    return 0;
}

// create new binder
int s_new_bind ( SymbolTableEntry* entry, Binding bind) {
    if(entry->type >= 0) {
        char * typename = s_table_short_type_name( entry->type );
        int tmpid = s_table_new_bindid();
        sprintf( bind, "%s_%s%d_s%d\0", entry->lex,typename, tmpid, entry->scope[1] );
    }
    else {
        sprintf( bind, "D_%s\0", entry->lex);
    }
    return 0;
}

int tmp_new_key ( Lexeme lex, ScopeId scope, SymbolTableKey key) {
    sprintf( key, "%s%d\0", lex, scope );
    return 0;
}

int tmp_new_bind ( SymbolTableEntry* entry, Binding bind) {
    sprintf( bind, "%s\0", entry->lex );
    return 0;
}

SymbolTableEntry* tmp_new_var_entry (Lexeme lex, int type, ScopeId sid ) {
    if( type==FUNC_T || type==FUNC_LITERAL_T ) {
        fprintf(stderr, "Hey wrong call to s_new_var_entry!\n");
        return NULL;
    }
    SymbolTableEntry* entry = (SymbolTableEntry*) malloc ( sizeof (SymbolTableEntry) );
    strcpy ((char *) entry->lex, lex);
    entry->line = -1;
    entry->type = type;
    entry->rtype = NOT_AVAIL;
    entry->typeCon = NULL;
    entry->scope[0] = -1;
    entry->scope[1] = sid;
    tmp_new_key ( entry->lex, entry->scope[1], entry->key );
    tmp_new_bind ( entry, entry->bind );
    return entry;
}

int s_entry_copy ( SymbolTableEntry * dest, SymbolTableEntry * source ) {
    strcpy( dest->lex, source->lex );
    dest->type = source->type;
    dest->rtype = source->rtype;
    dest->typeCon = NULL;
    dest->scope[0] = source->scope[0];
    dest->scope[1] = source->scope[1];
    strcpy( dest->key, source->key );
    strcpy( dest->bind, source->key );
    dest->line = source->line;
    return 0;
}

```

../src/SymbolTable.c

```

// author : Jing
#ifndef SYMBOLTABLEUTIL_H_NSBL_
#define SYMBOLTABLEUTIL_H_NSBL_
#include "ASTree.h"
int sTableDeclare(struct Node* node);
int sTableLookupId(struct Node* node);
SymbolTableEntry* sTableTryLookupId(struct Node* node);
int sTableLookupFunc(struct Node* node);
int sTableInsertTree(struct Node* node, int ttype);
int sTableInsertId(struct Node* node, int ttype);
int sTableInsertFunc(struct Node* node);
int sTableInsertFuncLiteral(struct Node* node);
void FuncHead(char* funcId, GArray* typeCon, FILE* out);
GArray* rmDynFromTypeCon(GArray* t);
int checkTwoTypeConsExceptDyn(GArray* t1, GArray* t2);

```

```
#endif
```

../src/SymbolTableUtil.h

```
/* author : Jing
#include <stdio.h>
#include <stdlib.h>
#include "global.h"
#include "SymbolTable.h"
#include "SymbolTableUtil.h"
#include "Error.h"

/** declare the variables or parameters */
int sTableDeclare(struct Node* node) {
    if(node->token == AST_DECLARATION ||
        node->token == AST_PARA_DECLARATION) { // var declaration
        int ttype = node->child[0]->lexval.ival;
        struct Node* nlist = node->child[1];
        // insert all IDENTIFIER as TYPE ttype in tree nlist
        sTableInsertTree(nlist, ttype);
    }
    if(node->token == AST_FUNC ) { // func declaration
        sTableInsertFunc(node);
    }
    if(node->token == FUNC_LITERAL ) { // func_literal
        sTableInsertFuncLiteral(node);
    }
    return 0;
}

/** insert all IDENTIFIER or DYN_ATTRIBUTE in the subtree */
int sTableInsertTree(struct Node* node, int ttype) {
    if(node == NULL) return;
    switch (node->token) {
        case IDENTIFIER :
            sTableInsertId(node, ttype); break;
        case DYN_ATTRIBUTE :
            // Do NOT insert attribute to symbol table
            //sTableInsertId(node, -ttype);
            break;
        case AST_COMMA :
            sTableInsertTree(node->child[0], ttype);
            sTableInsertTree(node->child[1], ttype); break;
        case AST_ASSIGN :
            sTableInsertTree(node->child[0], ttype); break;
        case BELONG:
            // Do NOT insert attribute to symbol table
            //sTableInsertTree(node->child[1], -ttype);
            break;
        default :
            fprintf(stderr, "sTableInsertTree : unknown token %d\n", node->token);
    }
    return 0;
}

/** insert one IDENTIFIER or DYN_ATTRIBUTE */
int sTableInsertId(struct Node* node, int ttype) {
    if ( ttype < DYN_STRING_T && ttype!=DYNAMIC_T) { // take care of declare unsupported type for attribute
        ERRNO = ErrorAttributeTypeNotSupported;
        errorInfo(ERRNO, node->line, "Type '%s' is not supported for Attribute\n", sTypeName(ttype));
        return ERRNO;
    }
    else if ( ttype == VOID_T ) {
        ERRNO = ErrorVoidTypeVariableNotSupported;
        errorInfo(ERRNO, node->line, " cannot declare 'void' variable.\n");
        return ERRNO;
    }
}

SymbolTableEntry* entry = sNewVarEty ( node->lexval.sval, ttype, node->line );
if ( sTableInsert( entry ) == ErrorSymbolTableKeyAlreadyExsit ) {
    SymbolTableEntry * te = sTableLookup(entry->key);
    ERRNO = ErrorIdentifierAlreadyDeclared;
    errorInfo(ERRNO, node->line, "%s% s' is already declared.\n", (ttype<0)?"@":"", node->lexval.sval);
    errorInfoNote("%s% s' is first declared at line %d\n",
        (ttype<0)?"@":"", node->lexval.sval, te->line);
    return ERRNO;
    // should tell where first declared
}
node->symbol = entry;
node->type = ttype;
```

```

    return 0;
}

/** insert a function */
int sTableInsertFunc(struct Node* node) {
    // declaration_specifiers : node->child[0]
    // func_declarator         : node->child[1]
    // func_id                 : node->child[1]->child[0]
    struct Node* declSpec = node->child[0];
    struct Node* funcId   = node->child[1]->child[0];
    SymbolTableEntry* entry = sNewFunEty ( funcId->lexval.sval, FUNC_T, declSpec->lexval.ival, node->typeCon, node->
        scope[0], node->scope[1], node->line );
    if ( sTableInsert( entry ) == ErrorSymbolTableKeyAlreadyExsit ) {
        SymbolTableEntry * te = sTableLookup(entry->key);
        ERRNO = ErrorIdentifierAlreadyDeclared;
        errorInfo(ERRNO, node->line, "%s' is already declared.\n", funcId->lexval.sval);
        errorInfoNote("%s' is first declared at line %d\n",
            funcId->lexval.sval, te->line);
        return ERRNO;
    }
    node->symbol = entry;
    node->type = FUNC_T;
    return 0;
}

/** insert a func_literal */
int sTableInsertFuncLiteral(struct Node* node) {
    struct Node* declSpec = node->child[1];
    struct Node* funcId   = node->child[0]->child[0];
    SymbolTableEntry* entry = sNewFunEty ( funcId->lexval.sval, FUNC_LITERAL_T, declSpec->lexval.ival, node->typeCon,
        node->scope[0], node->scope[1], node->line );
    if ( sTableInsert( entry ) == ErrorSymbolTableKeyAlreadyExsit ) {
        SymbolTableEntry * te = sTableLookup(entry->key);
        ERRNO = ErrorIdentifierAlreadyDeclared;
        errorInfo(ERRNO, node->line, "%s' is already declared.\n", funcId->lexval.sval);
        errorInfoNote("%s' is first declared at line %d\n",
            funcId->lexval.sval, te->line);
        return ERRNO;
    }
    node->symbol = entry;
    node->type = FUNC_T;
    return 0;
}

/** lookup an Id from symtable, if not found report compiling error */
int sTableLookupId(struct Node* node) {
    SymbolTableEntry* entry = sTableTryLookupId(node);
    if(entry == NULL) {
        ERRNO = ErrorIdentifierUsedBeforeDeclaration;
        errorInfo(ERRNO, node->line, "%s' is not declared before.\n", node->lexval.sval);
        return ERRNO;
    }
    node->symbol = entry;
    node->type = entry->type;
    return 0;
}

/** lookup an Id from symtable, return the entry */
SymbolTableEntry* sTableTryLookupId(struct Node* node) {
    if( node->token != IDENTIFIER &&
        node->token != DYN_ATTRIBUTE ) {
        fprintf(stderr, "error: sTableLookupId: argument must be IDENTIFIER or DYN_ATTRIBUTE\n");
        exit(EXIT_FAILURE);
    }
    SymbolTableKey key;
    SymbolTableEntry* entry;
    ScopeId id = sStackTopId;
    sStackReset();
    while (id >= 0) {
        sNewKey(node->lexval.sval, id, key);
        if ( (entry = sTableLookup(key)) != NULL ) break;
        id = sStackDown();
    }
    if (entry == NULL) {
        // disable error report here
        // printf("key= '%s'\n", key);
        // sTableShow(stderr);
    }
    return entry;
}

```

```

/** lookup a func or func_literal, if not found report compiling error */
int sTableLookupFunc(struct Node* node) {
    if (node->token != AST_FUNC_CALL) {
        fprintf(stderr, "error: sTableLookupFunc: argument must be AST_FUNC_CALL\n");
        exit(EXIT_FAILURE);
    }
    SymbolTableKey key;
    SymbolTableEntry* entry;
    struct Node* funcId = node->child[0];
    // create key
    sNewKey(funcId->lexval.sval, 0, key); // function always in scope 0
    // try lookup in Symbol Table
    entry = sTableLookup(key);
    if (entry == NULL) {
        ERRNO = ErrorFunctionCalledBeforeDeclaration;
        errorInfo(ERRNO, node->line, "");
        FuncHead(funcId->lexval.sval, node->typeCon, ERRORIO);
        errorInfoExt("' is not declared before.\n");
        return ERRNO;
    }
    // if found, check parameter types
    GArray* caller = node->typeCon;
    GArray* ref = entry->typeCon;
    int flag = 0;
    if (flag == ErrorFunctionCallNOTEqualNumberOfParameters ||
        flag == ErrorFunctionCallIncompatibleParameterType ||
        flag == ErrorFuncLiteralCallIncompatibleParameterType ) {
        errorInfoNote("function ");
        FuncHead(funcId->lexval.sval, ref, ERRORIO);
        errorInfoExt("' first declared at line %d\n", entry->line);
        return ERRNO;
    }
    // found correct one
    node->symbol = entry;
    node->type = entry->rtype;
    return 0;
}

/** output function heading */
void FuncHead(char* funcId, GArray* typeCon, FILE* out) {
    fprintf(out, "%s()", funcId);
    /*
    int i, ll=typeCon->len;
    for (i=0; i<ll-1; ++i)
        fprintf(out, "%s", sTypeName( g_array_index(typeCon, int, i) ));
    fprintf(out, "%s", sTypeName( g_array_index(typeCon, int, ll-1) ));
    */
}

/** check equivalence of two type constructors */
int checkTwoTypeCons(GArray* t1, GArray* t2) {
    if (t1->len!=t2->len) return 0;
    int i;
    for (i=0; i<t1->len; ++i) {
        int it1 = g_array_index(t1, int, i);
        int it2 = g_array_index(t2, int, i);
        if ( it1 >= 0 && it2 >= 0 && it1 != it2 ) // ignore dynamic type
            return 0;
    }
    return 1;
}

/** remove dynamic type from type constructors */
GArray* rmDynFromTypeCon(GArray* t) {
    int i;
    GArray* tga = g_array_new ( 1, 1, sizeof(int) );
    for (i=0; i<t->len; ++i) {
        int type = g_array_index(t, int, i);
        if (type>=0) g_array_append_vals ( tga, (gconstpointer) & type, 1);
    }
    return tga;
}

/** check equivalence of two type constructors, ignore dynamic types */
int checkTwoTypeConsExceptDyn(GArray* t1, GArray* t2) {
    GArray *ft1,*ft2;
    ft1 = rmDynFromTypeCon(t1);
    ft2 = rmDynFromTypeCon(t2);
    int rlt = checkTwoTypeCons(ft1,ft2);
    g_array_free(ft1,1);
    g_array_free(ft2,1);
}

```

```

    return rlt;
}

```

../src/SymbolTableUtil.c

4 Code Generator

```

// author : Jing , Lixing
#ifndef CODEGEN_H_NSBL_
#define CODEGEN_H_NSBL_
#include "ASTree.h"

#define REMOVE_DYN 0xF01

// node->tmp[]
#define GLOBAL_TMP 0
#define MATCH_TMP 1

int codeGen(struct Node * node);
void derivedTypeInitCode(struct Node* node, int type, int isglobal);
void stringInitCode(struct Node* node, int type, int isglobal);
void listInitCode(struct Node* node, int type, int isglobal);
int listCountCheck(struct Node* node, int type);
int codeAttr ( struct Node * node );
char * codeGetAttrVal( char * operand, int type, int lno );
char * codeFrontDecl(int lvl );
int codeAssignLeft( struct Node * node);
int codeFuncWrapDynArgs(struct Node* node, GArray* tcon, int* cnt);
char * codeForFreeDerivedVabInScope(ScopeId sid, int type, GList * gl, ScopeId lvl, int which);
char * codeForInitTmpVabInScope ( ScopeId sid, int type, GList * gl, ScopeId lvl, int which );
char * allFreeCodeInScope(ScopeId sid, GList * gl, ScopeId lvl);
char * allInitTmpVabCodeInScope(ScopeId sid, GList * gl, ScopeId lvl);
GList * getAllParaInFunc(struct Node * node, GList * gl);
GList * getReturnVab( struct Node * node, GList * gl);
GList * getAllScopeIdInside( struct Node * node, GList * gl, struct Node * target, int * rlt);

int codeAllGen(struct Node* node, char ** mainCode, char ** funCode);
void codeAllFuncLiteral(struct Node* node, char ** code);
void codeInclude(char ** code);
void codeAllGlobal(struct Node* node, char ** code);
char * wapperMainCode(char * mainBodyCode);
void exportCode(char * code);

#endif

```

../src/CodeGen.h

```

/*****
for author : see below
*****/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "CodeGen.h"
#include "SymbolTable.h"
#include "Parser.tab.h"
#include "global.h"
#include "Error.h"
#include "operator.h"
#include "CodeGenUtil.h"
#include "NSBLio.h"
#include "Derivedtype.h"

char * OUTFILE; // Output file name
FILE * OUTFILESTREAM; // Output file stream

int inLoop, // flags to indicate : inside of loop or func
inFunc, // : inside of func_literal
inFuncLiteral, // : inside of func_literal
isFunc; // : +1 : func / -1 : func_literal
int inMATCH, // flags to indicate : inside of match operator
existMATCH, // : exist match operator in subtree of AST
existPIPE, // : exist pipe operator in subtree of AST
nMATCHsVab; // count number of dynamic variables in Match

GList *returnList, *noReturn, *FuncParaList;

```

```

GList *returnList2, *noReturn2;
char * matchStaticVab, *frontDeclExp, *frontDeclExpTmp1;
char * LoopGotoLabel;
struct Node * FuncBody, *FuncLiteralBody, * LoopBody;

/** recursively generate code piece on each node */
int codeGen (struct Node * node) {
    if( node == NULL ) return;
    int token = node->token, errflag = 0;
    char * op = opMacro(token);
    struct Node *lf, *rt, *sg;
    char* printFunc;
    char* var;
    char* endBrace;
    char* printVattr;
    char* printCall;
    char* fileloc;
    char* comma;
    switch (token) {
/*****
        case INTEGER_CONSTANT :
            // AUTHOR : Jing and Lixing
        case FLOAT_CONSTANT :
        case BOOL_TRUE :
        case BOOL_FALSE :
        case STRING_LITERAL :
            // code and type already done in ASTree.c
            break;
        case IDENTIFIER :
            // type is done when insert into symtable
            if (node->symbol->bind!=NULL){ // should always true
                if (inMATCH==0){ // not in Match
                    node->code = strCatAlloc("", 1, node->symbol->bind);
                    if (node->type == VERTEX_T || node->type == EDGE_T ||
                        node->type == VLIST_T || node->type == ELIST_T
                        || node->type == STRING_T || node->type == GRAPH_T) {
                        node->codetmp = strCatAlloc("", 2, " ", node->symbol->bind);
                    }
                    else {
                        node->codetmp = strCatAlloc("", 1, node->symbol->bind);
                    }
                }
                else{ // in Match
                    node->code = strCatAlloc("", 3, "_str->", node->symbol->bind, "_match");
                    if (node->type == VERTEX_T || node->type == EDGE_T ||
                        node->type == VLIST_T || node->type == ELIST_T
                        || node->type == STRING_T || node->type == GRAPH_T) {
                        node->codetmp = strCatAlloc("", 2, " ", node->symbol->bind);
                    }
                    else {
                        node->codetmp = strCatAlloc("", 1, node->symbol->bind);
                    }
                }
                matchStaticVab = strRightCatAlloc( matchStaticVab, "", 5,
                    INDENT[1], sTypeName(node->type),
                    (node->type == VERTEX_T || node->type == EDGE_T ||
                     node->type == VLIST_T || node->type == ELIST_T
                     || node->type == STRING_T || node->type == GRAPH_T)
                     ? " * " : " ",
                    node->symbol->bind, "_match;\n");
                frontDeclExpTmp1 = strRightCatAlloc( frontDeclExpTmp1, "", 2,
                    (nMATCHsVab++==0) ? " " : " ",
                    node->symbol->bind);
            }
        }
        else
            ERRNO = ErrorNoBinderForId;
        break;
        case DYN_ATTRIBUTE :
            node->code = strCatAlloc("", 6,
                "object_get_attribute( _obj, _obj_type, ",
                "\":::", node->lexval.sval, "\", 0, ", strLine(node->line), " ) ");
            node->type = DYNAMIC_T;
            break;
        case AST_COMMA :
            lf = node->child[0]; rt = node->child[1];
            codeGen( lf ); codeGen( rt );
            node->code = strCatAlloc(" ", 3, lf->code, ",", rt->code);
            if (node->scope[0]==0){ // for global declaration
                node->codetmp = strCatAlloc(" ", 3, lf->codetmp, ",", rt->codetmp);
            }
    }
}

```

```

        node->type = node->child[1]->type;
        break;
    case AST_LIST_INIT:
        node->type = LIST_INIT_T;
        if(node->child==NULL)
            node->code = strCatAlloc("", 1, " ");
        else{
            sg = node->child[0];
            codeGen(sg);
            node->code = strCatAlloc("", 1, sg->code);
        }
        break;
    case AST_TYPE_SPECIFIER :
        node->code = strCatAlloc(" ",1,sTypeName(node->lexval.ival));
        break;
    case AST_DECLARATION : {
        lf = node->child[0];
        rt = node->child[1];
        codeGen( lf );
        int ttype = lf->lexval.ival;
        if( ttype != VLIST_T && ttype != ELIST_T )
            codeGen( node->child[1] );
        node->code = codeFrontDecl(node->scope[0] );
        // when the declaration is in scope 0, we need to generate two places of code for c
        // 1. external global declaration
        // 2. assignment in main func, if possible
        if(node->scope[0]==0) {
            switch(ttype){
                case GRAPH_T:
                case VERTEX_T:
                case EDGE_T:
                    derivedTypeInitCode(rt, ttype, 1);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                case STRING_T:
                    stringInitCode(rt, ttype, 1);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                case VLIST_T:
                case ELIST_T:
                    listInitCode(rt, ttype, 1);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                default:
                    node->code = strRightCatAlloc(node->code,"",3,INDENT[node->scope[0]],rt->code,";\n");
            }
            node->codetmp = strCatAlloc("",5,INDENT[node->scope[0]],lf->code," ",rt->codetmp,";\n");
        }
        // If scope > 0, no bother, just declaration everything in one c declaration
        else {
            switch(node->child[0]->lexval.ival){
                case GRAPH_T:
                case VERTEX_T:
                case EDGE_T:
                    derivedTypeInitCode(rt, ttype, 0);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                case STRING_T:
                    stringInitCode(rt, ttype, 0);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                case VLIST_T :
                case ELIST_T :
                    listInitCode(rt, ttype, 0);
                    node->code = strRightCatAlloc(node->code,"", 1, rt->code);
                    break;
                default:
                    node->code = strRightCatAlloc(node->code,"",5,INDENT[node->scope[0]],lf->code," ",rt->code,";\n");
            }
        }
        break;
    }
}
}
/*****/
case AST_ASSIGN : // AUTHOR : Jing
    if(inMATCH > 0) {
        ERRNO = ErrorAssignInMatch;
        errorInfo ( ERRNO, node->line, "assignment in Match operator.\n");
        return ERRNO;
    }
    lf = node->child[0]; rt = node->child[1];

```

```

if(lf->token != IDENTIFIER && lf->token != AST_ATTRIBUTE && lf->token != DYN_ATTRIBUTE) {
    ERRNO = ErrorAssignLeftOperand;
    errorInfo ( ERRNO, node->line, "the left operand of assign operator MUST be IDENTIFIER or ATTRIBUTE.\n"
    );
    return ERRNO;
}
codeAssignLeft(lf);
codeGen(rt);
// type check and implicit type conversion
if(lf->type == rt->type && lf->type >= 0 ) {
    if ( lf->type == INT_T || lf->type == FLOAT_T || lf->type == BOOL_T ) {
        node->code = strCatAlloc(" ",3,lf->code,op,rt->code);
        node->type = lf->type;
    }
    else if ( lf->type == STRING_T || lf->type == VLIST_T || lf->type == ELIST_T ||
        lf->type == VERTEX_T || lf->type == EDGE_T ||
        lf->type == GRAPH_T ){
        char * func = assignFunc(lf->type);
        node->type = lf->type;
        node->code = strCatAlloc(" ", 6,
            func, " ( &(", lf->code, " ) , ( ",
            rt->code, " ) ) "
        );
    }
    else {
        ERRNO = ErrorOperatorNotSupportedByType;
        errorInfo(ERRNO, node->line, "operator '%s' is not supported by type '%s'\n",op,sTypeName(lf->type)
        );
        return ERRNO;
    }
}
// float ==> int
else if (lf->type == INT_T && rt->type == FLOAT_T) {
    node->code = strCatAlloc(" ",4,lf->code,op,"(int)", rt->code);
    node->type = INT_T;
}
// int ==> float
else if (lf->type == FLOAT_T && rt->type == INT_T) {
    node->code = strCatAlloc(" ",4,lf->code,op,"(float)", rt->code);
    node->type = FLOAT_T;
}
// DYNAMIC
else if (lf->type < 0 || rt->type < 0) {
    if (lf->type < 0 ) { // DYNAMIC = DYNAMIC or STATIC
        int flag = 0;
        if (rt->type >=0) flag = codeAttr(rt);
        if (!flag){
            frontDeclExp = strRightCatAlloc(frontDeclExp, " ", 10,
                INDENT[node->scope[0]],
                "assign_operator (", lf->code, " , ", rt->code,
                (lf->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR",
                (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR",
                " , ", strLine(node->line), " );\n "
            );
            node->code = strCatAlloc(" ", 1, lf->code);
        }
        node->type = DYNAMIC_T;
        //node->tmp[0] = REMOVE_DYN;
    }
    else { // STATIC = DYNAMIC
        frontDeclExp = strRightCatAlloc(frontDeclExp, " ", 11,
            INDENT[node->scope[0]],
            "assign_operator_to_static (", rt->code, " , ", typeMacro(lf->type),
            " , (void *)&", lf->code,
            (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR",
            " , ", strLine(node->line), " );\n " );
        node->code = strCatAlloc(" ", 1, lf->code);
        //debugInfo("%s\n",node->code);
        node->type = lf->type;
    }
}
}
else { // ERROR
    node->code = NULL;
    ERRNO = ErrorTypeMismatch;
    errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n",op);
    return ERRNO;
}
// for global declaration in c
if(node->scope[0]==0){
    node->codetmp = strCatAlloc(" ",1,lf->codetmp);
}

```



```

        break;

/*****
case APPEND : // AUTHOR: Lixing
    lf = node->child[0]; rt = node->child[1];
    codeGen( lf );codeGen( rt );
    // TODO : DONE
    if(lf->type==GRAPH_T && rt->type==VERTEX_T){
        node->code = strCatAlloc( " , 5, "g_insert_v(", lf->code, " , ", rt->code, ")" );
    }
    else if(lf->type==GRAPH_T && rt->type==EDGE_T){
        node->code = strCatAlloc( " , 5, "g_insert_e(", lf->code, " , ", rt->code, ")" );
    }
    else if(lf->type==GRAPH_T && (rt->type==VLIST_T || rt->type == ELIST_T) ){
        node->code = strCatAlloc( " , 5, "g_append_list(", lf->code, " , ", rt->code, ")" );
    }
    else if(lf->type==VLIST_T && rt->type==VERTEX_T){
        node->code = strCatAlloc( " , 5, "list_append(", lf->code, " , VERTEX_T, " , rt->code, ")" );
    }
    else if(lf->type==ELIST_T && rt->type==EDGE_T){
        node->code = strCatAlloc( " , 5, "list_append(", lf->code, " , EDGE_T, " , rt->code, ")" );
    }
    else{
        ERRNO = ErrorAssignmentExpression;
        errorInfo(ERRNO, node->line, "append expression error\n");
        return ERRNO;
    }
    break;
*****/

/*****
case OR : // AUTHOR : Jing
case AND :
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf);codeGen(rt);
    node->type = BOOL_T;
    if(lf->type >= 0 && rt->type >= 0) {
        if (lf->type != rt->type){
            ERRNO = ErrorTypeMismatch;
            errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n",op);
            return ERRNO;
        }
        else if (lf->type == BOOL_T) {
            node->code = strCatAlloc( " ,3,lf->code,op,rt->code);
        }
        else {
            ERRNO = ErrorOperatorNotSupportedByType;
            errorInfo(ERRNO, node->line, "operator '%s' is only supported by type 'bool'\n",op);
            return ERRNO;
        }
    }
    else { // DYNAMIC
        int flag = 0;
        if (lf->type > 0 && lf->type == BOOL_T) flag = codeAttr(lf);
        if (rt->type > 0 && rt->type == BOOL_T) flag = codeAttr(rt);
        if (lf->type > 0 && lf->type != BOOL_T ||
            rt->type > 0 && rt->type != BOOL_T) {
            ERRNO = ErrorOperatorNotSupportedByType;
            errorInfo(ERRNO, node->line, "operator '%s' is only supported by type 'bool'\n",op);
            return ERRNO;
        }
        if(!flag) {
            SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
            frontDeclExp = strRightCatAlloc( frontDeclExp, " , 16,
                INDENT[node->scope[0]],
                "assign_operator_attr ( &( " , e->bind, " ) , " ,
                " binary_operator ( " , lf->code, " , ", rt->code, " , " , DynOP(token),
                " , FLAG_NO_REVERSE",
                ( lf->tmp[0]==REMOVE_DYN ) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ) ,
                ( rt->tmp[0]==REMOVE_DYN ) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ) ,
                " , " , strLine(node->line), " ) );\n"
            );
            node->code = strCatAlloc( " ,1,e->bind);
        }
        node->type = DYN_BOOL_T;
    }
    break;
*****/

/*****
case EQ : // AUTHOR : Jing
case NE :
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf);codeGen(rt);
    if(lf->type >= 0 && rt->type >= 0){ // STATIC

```

```

        if (lf->type != rt->type) {
            if (lf->type == INT_T && rt->type == FLOAT_T) {
                node->code = strCatAlloc(" ", 4, "(float)", lf->code, op, rt->code);
                node->type = BOOL_T;
            }
            else if (lf->type == FLOAT_T && rt->type == INT_T) {
                node->code = strCatAlloc(" ", 4, lf->code, op, "(float)", rt->code);
                node->type = BOOL_T;
            }
            else {
                ERRNO = ErrorTypeMismatch;
                errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n", op);
                return ERRNO;
            }
        }
        else {
            node->code = strCatAlloc(" ", 3, lf->code, op, rt->code);
            node->type = BOOL_T;
        }
    }
    else { // DYNAMIC
        int flag = 0;
        if (lf->type >= 0) flag = codeAttr(lf);
        if (rt->type >= 0) flag = codeAttr(rt);
        if (!flag) {
            SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 15,
            INDENT[node->scope[0]],
            "assign_operator_attr( &( ", e->bind,
            " ), binary_operator ( ", lf->code, " , ", rt->code, " , ", DynOP(token),
            " , FLAG_NO_REVERSE",
            ( (lf->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ),
            ( (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ),
            " , ", strLine(node->line), " ) );\n"
            );
            node->code = strCatAlloc("", 1, e->bind);
        }
        node->tmp[0] = REMOVE_DYN;
        node->type = DYN_BOOL_T;
    }
    break;
}
/*****
case LT : // AUTHOR : Jing
case GT :
case LE :
case GE :
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf); codeGen(rt);
    node->type = BOOL_T;
    if (lf->type == rt->type && (lf->type == INT_T) || (lf->type == FLOAT_T) )
        node->code = strCatAlloc(" ", 3, lf->code, op, rt->code);
    else if (lf->type == INT_T && rt->type == FLOAT_T)
        node->code = strCatAlloc(" ", 4, "(float)", lf->code, op, rt->code);
    else if (rt->type == INT_T && lf->type == FLOAT_T)
        node->code = strCatAlloc(" ", 4, lf->code, op, "(float)", rt->code);
    else if (lf->type <= 0 || rt->type <= 0) { // DYNAMIC
        if (lf->type >= 0 && lf->type != INT_T && lf->type != FLOAT_T ||
            lf->type >= 0 && lf->type != INT_T && lf->type != FLOAT_T ){
            ERRNO = ErrorTypeMismatch;
            errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n", op);
            return ERRNO;
        }
        int flag = 0;
        if (lf->type >= 0) flag = codeAttr(lf);
        if (rt->type >= 0) flag = codeAttr(rt);
        if (!flag) {
            SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 15,
            INDENT[node->scope[0]],
            "assign_operator_attr( &( ", e->bind,
            " ), binary_operator ( ", lf->code, " , ", rt->code, " , ", DynOP(token),
            " , FLAG_NO_REVERSE",
            ( (lf->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ),
            ( (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ),
            " , ", strLine(node->line), " ) );\n"
            );
            node->code = strCatAlloc("", 1, e->bind);
        }
        node->type = DYNAMIC_T;
    }
}
else {

```

```

        ERRNO = ErrorTypeMismatch;
        errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n",op);
        return ERRNO;
    }

    break;
/*****
case ADD : // AUTHOR : Jing
case SUB :
case MUL :
case DIV :
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf);codeGen(rt);
    if(lf->type == rt->type && (lf->type == INT_T) || (lf->type == FLOAT_T) ) {
        node->code = strCatAlloc(" ",3,lf->code,op,rt->code);
        node->type = lf->type;
    }
    else if (lf->type == INT_T && rt->type == FLOAT_T) {
        node->code = strCatAlloc(" ",4, "(float)",lf->code,op,rt->code);
        node->type = FLOAT_T;
    }
    else if (rt->type == INT_T && lf->type == FLOAT_T) {
        node->code = strCatAlloc(" ",4, lf->code,op,"(float)",rt->code);
        node->type = FLOAT_T;
    }
    else if (lf->type < 0 || rt->type < 0) { // DYNAMIC
#ifdef _DEBUG
        debugInfo("DYNAMIC : %d : (%d, %d) \n",
            node->token, lf->type, rt->type );
#endif

        int flag = 0;
        if(lf->type>=0) flag = codeAttr(lf); // if STATIC, wapper to Attr
        if(rt->type>=0) flag = codeAttr(rt);
        if (!flag) {
            SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
            frontDeclExp = strRightCatAlloc( frontDeclExp, " ", 15,
                INDENT[node->scope[0]],
                "assign_operator_attr( &( ", e->bind,
                " ) , binary_operator( ", lf->code, " , ", rt->code, " , ", DynOP(token),
                " , FLAG_NO_REVERSE",
                (lf->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ,
                (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR" ,
                " , ", strLine(node->line), " ) );\n"
            );
            node->code = strCatAlloc(" ", 1, e->bind);
            node->type = DYNAMIC_T;
        }
    }
    else {
        ERRNO = ErrorTypeMismatch;
        errorInfo(ERRNO, node->line, "type mismatch for the operands of operator '%s'\n",op);
        return ERRNO;
    }
    break;
/*****
case AST_CAST : // AUTHOR : Jing
    lf = node->child[0]; rt = node->child[1];
    int castType = lf->lexval.ival;
    codeGen(rt);
    if(rt->type >= 0) {
        if(castType == rt->type) {
            node->code = strCatAlloc(" ",4,"(",sTypeName(lf->lexval.ival),")" , rt->code);
            node->type = castType;
        }
        else if ( (castType == INT_T && rt->type == FLOAT_T) ||
            (castType == FLOAT_T && rt->type == INT_T) ) {
            node->code = strCatAlloc(" ",4,"(",sTypeName(lf->lexval.ival),")" , rt->code);
            node->type = castType;
        }
        else {
            ERRNO = ErrorCastType;
            errorInfo(ERRNO, node->line, "cast from '%s' to '%s' is invalid\n", sTypeName(castType), sTypeName(
                rt->type) );
            return ERRNO;
        }
    }
    else { // DYNAMIC
        SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
        frontDeclExp = strRightCatAlloc( frontDeclExp, " ",11 ,
            INDENT[node->scope[0]],
            "assign_operator_attr( &( ", e->bind,

```

```

        " ) , cast_operator( " , rt->code, " , " , typeMacro(castType),
        (rt->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR",
        " , " , strLine(node->line), " ) );\n"
    );
    node->code = strCatAlloc( " " , 1, e->bind);
    node->type = DYNAMIC_T;
}
break;
/*****
case AST_UNARY_PLUS : // AUTHOR : Jing
case AST_UNARY_MINUS :
case AST_UNARY_NOT :
    sg = node->child[0];
    codeGen(sg);
    if ( sg->type >= 0 ) {
        if ( (sg->type == INT_T || sg->type == FLOAT_T) &&
            ( token == AST_UNARY_PLUS || token == AST_UNARY_MINUS ) ) {
            node->code = strCatAlloc( " " , 4, op, "(" , sg->code, ")" );
            node->type = sg->type;
        }
        else if ( sg->type == BOOL_T && token == AST_UNARY_NOT ) {
            node->code = strCatAlloc( " " , 4, op, "(" , sg->code, ")" );
            node->type = sg->type;
        }
        else {
            ERRNO = ErrorOperatorNotSupportedByType;
            errorInfo(ERRNO, node->line, "unary operator '%s' is not supported by type '%s'.\n", op, sTypeName(sg
            ->type));
            return ERRNO;
        }
    }
    else { // DYNAMIC
        SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
        frontDeclExp = strRightCatAlloc( frontDeclExp, " " , 11,
            INDENT[node->scope[0]],
            "assign_operator_attr( &( " , e->bind,
            " ) , unary_operator ( " , sg->code, " , " , DynOP(token),
            (sg->tmp[0]==REMOVE_DYN) ? " , FLAG_DESTROY_ATTR" : " , FLAG_KEEP_ATTR",
            " , " , strLine(node->line), " ) );\n"
        );
        node->code = strCatAlloc( " " , 1, e->bind);
        if(token==AST_UNARY_NOT) node->type = DYN_BOOL_T;
        else node->type = DYNAMIC_T;
    }
    break;
/*****
case ARROW : // AUTHOR : Lixing
    lf = node->child[0]; sg = node->child[1]; rt = node->child[2];
    if(lf->token!=IDENTIFIER || sg->token!=IDENTIFIER || rt->token!=IDENTIFIER){
        ERRNO = ErrorEdgeAssignExpression;
        errorInfo(ERRNO, node->line, "edge assign expression error\n");
    }
    if(lf->type!=EDGE_T || sg->type!=VERTEX_T || rt->type!=VERTEX_T){
        ERRNO = ErrorEdgeAssignExpression;
        errorInfo(ERRNO, node->line, "edge assign illegal var type error\n");
    }
    codeGen(lf); codeGen(sg); codeGen(rt);
    node->code = strCatAlloc( " " , 7, "edge_assign_direction(" , lf->code, " , " , sg->code, " , " , rt->code, ")" );
    node->codetmp = NULL;
    break;
/*****
case AST_FUNC_CALL : // AUTHOR : Jing
    // lookup symbol table, also set type
    errflag = sTableLookupFunc(node);
    // code Gen
    if(!errflag) {
        if (node->nch > 1){ // if have args
            int cnt = 0; // count number of args
            codeFuncWrapDynArgs(node->child[1], node->symbol->typeCon, &cnt);
            if (node->symbol->typeCon->len != cnt) {
                ERRNO = ErrorFunctionCallNOTEqualNumberOfParameters;
                errorInfo(ERRNO, node->line, "function Call has inconsistent number of arguments to its
                declaration. %d, %d\n", node->symbol->typeCon->len, cnt);
            }
        }
        if(node->symbol->type == FUNC_LITERAL_T && inMATCH > 0) {
            if(node->nch == 1)
                node->code = strCatAlloc( " " , 2, node->symbol->bind, " ( _obj, _obj_type )" );
            else
                node->code = strCatAlloc( " " , 4, node->symbol->bind, " ( _obj, _obj_type, " ,
                node->child[1]->code, " )" );

```

```

    }
    else if (node->symbol->type == FUNC_T) {
        if (node->nch == 1) node->code = strCatAlloc(" ", 2, node->symbol->bind, "()");
        else node->code = strCatAlloc(" ", 4, node->symbol->bind, "(", node->child[1]->code, " ");
    }
    else {
        ERRNO = ErrorWrongFuncCall;
        errorInfo(ERRNO, node->line, "invalid func call.\n");
    }
}
break;
case AST_ARG_EXPS :
    codeGen(node->child[0]);
    node->type = node->child[0]->type;
    node->code = strCatAlloc(" ", 1, node->child[0]->code );
    break;
/*****
case PIPE :{
    // AUTHOR : Jing , Lixing
    lf = node->child[0];
    rt = node->child[1];
    codeGen(lf); codeGen(rt);
    if (lf->type != ELIST_T && lf->type != VLIST_T) {
        ERRNO = ErrorPipeWrongType;
        errorInfo(ERRNO, node->line, "pipe can NOT be operated on type '%s'.\n", sTypeName(lf->type));
    }
    existPIPE = 1;
    char* nltype = (lf->type == VLIST_T) ? typeMacro(EDGE_T) : typeMacro(VERTEX_T);
    SymbolTableEntry* enl = tmpVab( (lf->type == VLIST_T) ? ELIST_T : VLIST_T, node->scope[1] );
    SymbolTableEntry* elen = tmpVab( INT_T, node->scope[1] );
    SymbolTableEntry* ei = tmpVab( INT_T, node->scope[1] );
    char * cass = tmpVabAssign( enl, " new_list () " );
    char * ident = INDENT[node->scope[0]];
    frontDeclExp = strRightCatAlloc( frontDeclExp, "", 29,
        ident, "// START_PIPE\n",
        ident, cass,
        ident, enl->bind, "->type = ", nltype, ";\n",
        ident, "int ", elen->bind, " = g_list_length(", lf->code, "->list);\n",
        ident, "int ", ei->bind, ";\n",
        ident, "for(", ei->bind, "=0; ", ei->bind, "<", elen->bind, "; ", ei->bind, "++){ \n";
    if (lf->type == ELIST_T) {
        if (rt->token == STARTING_VERTICES)
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 8,
                enl->bind, " = list_append( ", enl->bind, ", VERTEX_T, ((EdgeType*)g_list_nth_data(", lf->code,
                    "->list, ", ei->bind, "))->start);\n");
        else if (rt->token == ENDING_VERTICES)
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 8,
                enl->bind, " = list_append( ", enl->bind, ", VERTEX_T, ((EdgeType*)g_list_nth_data(", lf->code,
                    "->list, ", ei->bind, "))->end);\n");
        else {
            // should not arrived here
        }
    }
    else if (lf->type == VLIST_T) {
        if (rt->token == OUTCOMING_EDGES)
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 8,
                enl->bind, " = list_append_gl(", enl->bind, ", ((VertexType*)g_list_nth_data(", lf->code, "->
                    list, ", ei->bind, "))->outEdges, EDGE_T);\n");
        else if (rt->token == INCOMING_EDGES)
            frontDeclExp = strRightCatAlloc( frontDeclExp, "", 8,
                enl->bind, " = list_append_gl(", enl->bind, ", ((VertexType*)g_list_nth_data(", lf->code, "->
                    list, ", ei->bind, "))->inEdges, EDGE_T);\n");
        else {
            // should not arrived here
        }
    }
    frontDeclExp = strRightCatAlloc( frontDeclExp, "", 1, "}" // END_PIPE\n");
    node->code = strCatAlloc(" ", 1, enl->bind);
    if (lf->type == ELIST_T)
        node->type = VLIST_T;
    else
        node->type = ELIST_T;
    break;
}
/*****
case AST_MATCH :
    // AUTHOR : Jing
    lf = node->child[0]; // list
    rt = node->child[1]; // condition
    sg = node->child[2]; // scope_out
    codeGen(lf);
    // get the STR name, func name,
    char * tmpfunc = tmpMatch();

```

```

char * match_str = tmpMatchStr();
char * match_str_val = tmpMatchStrVab();
// declaration of STR
frontDeclExpTmp1 = frontDeclExp; // store everything before match
frontDeclExp = NULL; // clear front code
frontDeclExpTmp1 = strRightCatAlloc( frontDeclExpTmp1, "", 5,
    "struct ", match_str, " ", match_str_val, " = {"
);
nMATCHsVab = 0;
inMATCH++; codeGen(rt); inMATCH--;
frontDeclExpTmp1 = strRightCatAlloc( frontDeclExpTmp1, "", 1, "};\n" );
if(lf->type != VLIST_T && lf->type != ELIST_T) {
    ERRNO = ErrorMatchWrongType;
    errorInfo(ERRNO,node->line," match can NOT be operated on type '%s'.\n",sTypeName(lf->type) );
    return ERRNO;
}
// check return type == bool
if(rt->type != BOOL_T && rt->type >=0 ){
    ERRNO = ErrorInvalidReturnType;
    errorInfo(ERRNO,node->line,"the body of Match operator must return bool result.\n");
    return ERRNO;
}
// set FLAG for STR declaration
// FLAG cleared in AST_EXP_STAT
existMATCH = 1;
if(rt->type < 0) { // if DYNAMIC, convert to BOOL_T
    char * ctmp = rt->code;
    rt->code = codeGetAttrVal( rt->code, BOOL_T, node->line );
    free(ctmp);
}
// first generate struct and func for this match
char* func_body = codeFrontDecl( 1 ); // get func body
char* freecode = allFreeCodeInScope( sg->scope[1], NULL, 1 );
char* initcode = allInitTmpVabCodeInScope( sg->scope[1], NULL, 1 );
SymbolTableEntry* ert = tmpVab( BOOL_T, sg->scope[1] );
node->codetmp = strCatAlloc("", 24,
    "struct ",match_str, " {\n",
    matchStaticVab,
    "};\n",
    "bool ", tmpfunc,
    " ( void * _obj, int _obj_type, struct ", match_str, " * _str ) {\n",
    initcode,
    func_body,
    INDENT[1], "bool ", ert->bind, " = ", rt->code, ";\n",
    freecode,
    INDENT[1], "return ", ert->bind, ";\n",
    "} // END_MATCH_FUNC \n"
);
free(func_body); free(freecode);free(initcode);
free(matchStaticVab); matchStaticVab =NULL;
frontDeclExp = frontDeclExpTmp1; // clear str decl body
// restore front code before match
//
int ttype = ( lf->type == VLIST_T )? VERTEX_T: EDGE_T ;
SymbolTableEntry* elt = tmpVab( VLIST_T, node->scope[1] );
SymbolTableEntry* elen = tmpVab( INT_T, node->scope[1] );
SymbolTableEntry* ei = tmpVab( INT_T, node->scope[1] );
SymbolTableEntry* eb = tmpVab( BOOL_T, node->scope[1] );
SymbolTableEntry* eobj = tmpVab( ttype, node->scope[1] );
char * cass = tmpVabAssign( elt, " new_list ()" );
char * ident = INDENT[node->scope[0]];
char * cdel = tmpVabDel( eobj );
frontDeclExp = strRightCatAlloc(frontDeclExp,"", 69,
    ident,"// START_MATCH\n",
    ident,cass,
    ident,elt->bind, "->type = (", lf->code, ")>type;\n",
    ident,"int ", elen->bind, " = g_list_length( (", lf->code, ")>list );\n",
    ident,"int ", ei->bind, ";\n",
    ident,"bool ", eb->bind, ";\n",
    ident,"for (", ei->bind, "=0; ", ei->bind, "<", elen->bind, "; ", ei->bind, "++) {\n",
    ident,assignFunc(ttype)," ( &( ", eobj->bind, " ), list_getelement ( ",
    lf->code, " , ", ei->bind, " );\n",
    ident,"if (", eb->bind, " = ", tmpfunc, " ( ",eobj->bind, " , ( ", lf->code, ")>type, &", match_str_val,
    " ) ) {\n",
    ident,elt->bind, " = list_append ( ", elt->bind, " , ", typeMacro(ttype)," , ", eobj->bind, " );\n",
    ident,"}\n",
    ident,cdel,
    ident,"} // END_MATCH \n"
);
node->code = strCatAlloc("",1,elt->bind);
node->type = lf->type;

```

```

        break;
/*****/
case AST_LIST_MEMBER: // AUTHOR : Jing
    lf = node->child[0];
    rt = node->child[1];
    codeGen(lf); codeGen(rt);
    if(lf->type != VLIST_T && lf->type != ELIST_T) {
        ERRNO = ErrorGetMemberForNotListType;
        errorInfo( ERRNO, node->line, "get member for not list type.\n");
        return ERRNO;
    }
    if(rt->type == INT_T){
        node->code = strCatAlloc( " ", 6 ,
            (lf->type == VLIST_T) ? "(VertexType *)" : "(EdgeType *)" ,
            "list_getelement ( ", lf->code, " , ", rt->code, " )" );
    }
    else if (rt->type < 0) { // DYNAMIC
        node->code = strCatAlloc( " ", 8,
            (lf->type == VLIST_T) ? "(VertexType *)" : "(EdgeType *)" ,
            "list_getelement ( ", lf->code,
            " , get_attr_value_INT_T ( ", rt->code, " , ", strLine(node->line), " ) )" );
    }
    node->type = (lf->type == VLIST_T) ? VERTEX_T : EDGE_T;
    break;
case AST_LENGTH:
    sg = node->child[0];
    codeGen(sg);
    if(sg->type != VLIST_T && sg->type != ELIST_T) {
        ERRNO = ErrorGetLengthForTypeNotList;
        errorInfo( ERRNO, node->line, "get length for type not list.\n");
        return ERRNO;
    }
    node->code = strCatAlloc( " ", 3,
        "g_list_length( ", sg->code, "->list )" );
    node->type = INT_T;
    break;
/*****/
case AST_ATTRIBUTE : // AUTHOR : Jing
    if(inMATCH==0){
        node->child[0]->code = strCatAlloc(" ", 1, node->child[0]->symbol->bind);
    }
    else {
        node->child[0]->code = strCatAlloc(" ", 3, "_str->", node->child[0]->symbol->bind, "_match");
        matchStaticVab = strRightCatAlloc( matchStaticVab, " ", 5,
            INDENT[1], sTypeName(node->child[0]->symbol->type),
            " * ", node->child[0]->symbol->bind, "_match;\n");
        frontDeclExpTmp1 = strRightCatAlloc( frontDeclExpTmp1, " ", 2,
            (nMATCHsVab++==0) ? " " : " , ",
            node->child[0]->symbol->bind);
    }
    node->child[1]->code = strCatAlloc(" ", 1, node->child[1]->lexval.sval);
    if(node->child[0]->type == VERTEX_T )
        node->code = strCatAlloc(" ", 7, "vertex_get_attribute( ",
            node->child[0]->code, " , \"", node->child[1]->code, "\", 0, ", strLine(node->line), ")");
    else if (node->child[0]->type == EDGE_T )
        node->code = strCatAlloc(" ", 7, "edge_get_attribute( ",
            node->child[0]->code, " , \"", node->child[1]->code, "\", 0, ", strLine(node->line), ")");
    else {
        ERRNO = ErrorGetAttrForWrongType;
        errorInfo(ERRNO, node->line, "Access attribute for type '%s'.\n",
            sTypeName(node->child[0]->type) );
        node->code = NULL;
    }
    node->type = DYN_ATTR_T;
    break;
/*****/
case AST_GRAPH_PROP : // AUTHOR : Lixing
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf); codeGen(rt);
    if(lf->type != GRAPH_T){
        ERRNO = ErrorWrongArgumentType;
        errorInfo(ERRNO, node->line, "need a graph type for AllV and AllE operation, but type used is '%s'. \n",
            sTypeName(lf->type) );
        return;
    }
    switch(rt->token){
        case ALL_VERTICES:
            node->type = VLIST_T;
            node->code = strCatAlloc(" ", 3, "get_g_vlist( ", lf->code, " );");
            break;

```

```

        case ALL_EDGES:
            node->type = ELIST_T;
            node->code = strCatAlloc("", 3, "get_g_elist(", lf->code, ")");
            break;
        default:
            ERRNO = ErrorOperatorNotSupportedByType;
            errorInfo(ERRNO, node->line, "Undifined Operation for graph \n");
            return;
    }
    break;
/*****
case AST_COMP_STAT : // AUTHOR : Jing
case AST_COMP_STAT_NO_SCOPE :
    if(node->nch == 0) { // empty
        node->code = strCatAlloc("", 2, INDENT[node->scope[0]], "{ } // EMPTY_COMP \n");
    }
    else {
        sg = node->child[0];
        codeGen(sg);
        char * freecode = NULL;
        char * initcode = NULL;
        if(token == AST_COMP_STAT) { // GC
            freecode = allFreeCodeInScope(node->child[1]->scope[1], NULL, node->child[1]->scope[0] );
            initcode = allInitTmpVabCodeInScope( node->child[1]->scope[1], NULL, node->child[1]->scope[0] );
            node->code = strCatAlloc("", 7, INDENT[node->scope[0]], "{\n", initcode,
                node->child[0]->code, freecode, INDENT[node->scope[0]], "}" // END_COMP\n");
        }
        else {
            node->code = strCatAlloc("", 5, INDENT[node->scope[0]], "// BEGIN_COMP_NO_SCOPE\n",
                node->child[0]->code, INDENT[node->scope[0]], "// END_COMP_NO_SCOPE\n");
        }
    }
    break;
case AST_STAT_LIST :
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf); codeGen(rt);
    node->code = strCatAlloc("", 2, lf->code, rt->code);
    break;
/*****
case AST_EXP_STAT : // AUTHOR : Lixing
    if(node->nch == 0) { // empty
        node->code = strCatAlloc("", 1, ";\n");
    }
    else {
        codeGen(node->child[0]);
        node->code = codeFrontDecl(node->scope[0]);
        node->code = strRightCatAlloc(node->code, "", 3, INDENT[node->scope[0]], node->child[0]->code, ";\n");
    }
    break;
/*****
case AST_IF_STAT : // AUTHOR : Jing
    lf = node->child[0]; rt = node->child[1];
    codeGen(lf);
    node->code = codeFrontDecl(node->scope[0]);
    if(lf->type == BOOL_T) {
        codeGen(rt);
        node->code = strRightCatAlloc(node->code, "", 7,
            INDENT[node->scope[0]], "if ( ", lf->code, " ){ \n",
            rt->code,
            INDENT[node->scope[0]], " } // END_IF \n");
    }
    else if (lf->type < 0) { // DYNAMIC
        SymbolTableEntry* etmp = tmpVab( DYN_ATTR_T, node->scope[1] );
        char * cassign = tmpVabAssign( etmp, lf->code );
        codeGen(rt);
        node->code = strRightCatAlloc(node->code, "", 11,
            INDENT[node->scope[0]], "// START_IF\n",
            INDENT[node->scope[0]], cassign,
            INDENT[node->scope[0]], "if ( ", codeGetAttrVal(etmp->bind, BOOL_T, node->line), " ) {\n",
            rt->code,
            INDENT[node->scope[0]], " } // END_IF\n"
        );
        free(cassign);
    }
    else {
        ERRNO = ErrorIfConditionNotBOOL;
        errorInfo(ERRNO, node->line, "condition in IF statement is NOT of type 'bool'.\n");
        return ERRNO;
    }
    break;
case AST_IFELSE_STAT :

```



```

lf = node->child[0]; sg = node->child[1]; rt = node->child[2];
codeGen(lf);
node->code = codeFrontDecl(node->scope[0]);
if(lf->type == BOOL_T) {
    codeGen(sg); codeGen(rt);
    node->code = strRightCatAlloc(node->code, "", 10,
        INDENT[node->scope[0]], "if ( ", lf->code, " ){ \n",
        sg->code,
        INDENT[node->scope[0]], "\nelse{\n", rt->code,
        INDENT[node->scope[0]], " } // END_IF\n");
}
else if (lf->type < 0) { // DYNAMIC
    SymbolTableEntry* etmp = tmpVab( DYN_ATTR_T, node->scope[1] );
    char * cassign = tmpVabAssign( etmp, lf->code );
    codeGen(sg); codeGen(rt);
    node->code = strRightCatAlloc(node->code, "", 14,
        INDENT[node->scope[0]], "// START_IF\n",
        INDENT[node->scope[0]], cassign,
        INDENT[node->scope[0]], "if ( ", codeGetAttrVal(etmp->bind, BOOL_T, node->line), " ) {\n",
        sg->code,
        INDENT[node->scope[0]], "} else {\n", rt->code,
        INDENT[node->scope[0]], " } // END_IF\n"
    );
    free(cassign);
}
else {
    ERRNO = ErrorIfConditionNotBOOL;
    errorInfo(ERRNO, node->line, "condition in IF statement is NOT of type 'bool'.\n");
    return ERRNO;
}
break;
}
/*****
case AST_WHILE : { // AUTHOR : Lixing
    lf = node->child[0]; rt = node->child[1];
    char * tmpcode;
    char * label = strCatAlloc("", 1, gotolabel());
    frontDeclExpTmp1 = frontDeclExp; frontDeclExp = NULL;
    codeGen(lf); tmpcode = frontDeclExp;
    frontDeclExp = frontDeclExpTmp1; frontDeclExpTmp1 = NULL;
    node->code = codeFrontDecl(node->scope[0] );
    inLoop++;
    LoopBody = ( rt->nch == 0 ) ? NULL: rt->child[0];
    LoopGotoLabel = label;
    codeGen(rt);
    LoopBody = NULL; LoopGotoLabel = NULL;
    inLoop--;

    if(lf->type>=0){
        node->code = strRightCatAlloc(node->code, "", 10,
            INDENT[node->scope[0]], "while ( ", lf->code, " ) {\n",
            rt->code, INDENT[node->scope[0]],
            label, ": { } ", INDENT[node->scope[0]],
            " } //END_OF_WHILE\n");
    }
    else { // DYNAMIC
        SymbolTableEntry* etmp = tmpVab( DYN_ATTR_T, node->scope[1] );
        char * cass = tmpVabAssign( etmp, lf->code );
        node->code = strRightCatAlloc(node->code, "", 20,
            INDENT[node->scope[0]], tmpcode,
            INDENT[node->scope[0]], "// START_OF_WHILE\n",
            INDENT[node->scope[0]], cass,
            INDENT[node->scope[0]], "while ( ", codeGetAttrVal(etmp->bind, BOOL_T, node->line),
            " ) {\n", rt->code,
            INDENT[node->scope[0]], label, ": {\n",
            INDENT[node->scope[0]], tmpcode,
            INDENT[node->scope[0]], cass,
            INDENT[node->scope[0]], "\n} //END_OF_WHILE\n");
        free(label); label = NULL;
        free(cass); cass = NULL;
    }
    free(tmpcode); tmpcode = NULL;
    free(label); label = NULL;
    break;
}
case AST_FOR : {
    struct Node *f1 = node->child[0],
        *f2 = node->child[1],
        *f3 = node->child[2],
        *fs = node->child[3];
    char * cf1 = NULL, *cf2 = NULL, *cf3 = NULL;
    char * label = strCatAlloc("", 1, gotolabel());

```

```

frontDeclExpTmp1 = frontDeclExp; frontDeclExp = NULL;
codeGen(f1); cf1 = frontDeclExp; frontDeclExp = NULL;
codeGen(f2); cf2 = frontDeclExp; frontDeclExp = NULL;
codeGen(f3); cf3 = frontDeclExp; frontDeclExp = NULL;
frontDeclExp = frontDeclExpTmp1; frontDeclExpTmp1 = NULL;
node->code = codeFrontDecl(node->scope[0] );
inLoop++;
LoopBody = ( fs->nch == 0 ) ? NULL: fs->child[0];
LoopGotoLabel = label;
codeGen(fs);
LoopBody = NULL; inLoop--; LoopGotoLabel = NULL;
if (f1->type>=0 && f2->type>=0 && f3->type>=0){
    node->code = strRightCatAlloc(node->code, "", 13, INDENT[node->scope[0]],
        "for (", (f1!=NULL)? f1->code : "", ";",
        (f2!=NULL)? f2->code : "", ";",
        (f3!=NULL)? f3->code : "", ") {\n",
        fs->code,
        INDENT[node->scope[0]],label,":{}\n",
        "} //END_OF_FOR\n");
}
else { // DYNAMIC :: translate for to while
    SymbolTableEntry* etmp = tmpVab( DYN_ATTR_T, node->scope[1] );
    char * cass = tmpVabAssign(etmp, f2->code);
    node->code = strRightCatAlloc(node->code, "", 28,
        INDENT[node->scope[0]], cf1, "\n", cf2, "\n",
        INDENT[node->scope[0]], "// START_OF_FOR\n",
        INDENT[node->scope[0]], cass,
        INDENT[node->scope[0]], "while (", codeGetAttrVal(etmp->bind, BOOL_T, node->line),
        ") {\n", fs->code,
        INDENT[node->scope[0]], label, ": {\n",
        INDENT[node->scope[0]], cf3, "\n",
        INDENT[node->scope[0]], cf2, "\n",
        INDENT[node->scope[0]], cass, ";\n",
        "}\n} \n",
        "//END_OF_FOR\n"
    );
    free(cf1);cf1=NULL;
    free(cf2);cf2=NULL;
    free(cf3);cf3=NULL;
    free(cass);cass=NULL;
}
free(label);
break;
}
case AST_FOREACH :{
    lf = node->child[0]; sg = node->child[1]; rt = node->child[2];
    // break or continue is forbidden
    codeGen(lf);
    codeGen(sg);
    node->code = codeFrontDecl(node->scope[0] );
    codeGen(rt);
    int ltype = lf->child[1]->type, rtype = sg->type;
    if( ltype==VERTEX_T&&rtype==VLIST_T || ltype==EDGE_T&&rtype==ELIST_T ){
        char* ti = lf->child[1]->symbol->bind;
        char* tlen = strCatAlloc("", 1, tmpVab(INT_T, node->scope[1]));
        char* tc = strCatAlloc("", 1, tmpVab(INT_T, node->scope[1]));
        node->code = strRightCatAlloc(node->code, "", 25,
            INDENT[node->scope[0]], "// START_FOREACH\n",
            INDENT[node->scope[0]], ti, " = NULL;\n",
            INDENT[node->scope[0]], "int ", tlen, " = g_list_length(", sg->code, "->list);\n",
            INDENT[node->scope[0]], "int ", tc, ";\n",
            INDENT[node->scope[0]], "for (", tc, "=0; ", tc, "<", tlen, "; ", tc, "++) {\n");
        if(ltype == VERTEX_T)
            node->code = strRightCatAlloc(node->code, "", 8,
                INDENT[node->scope[0]], "assign_operator_vertex(&", ti, ", g_list_nth_data ( ", sg->code, "->list, ", tc, " ) );\n");
        else
            node->code = strRightCatAlloc(node->code, "", 8,
                INDENT[node->scope[0]], "assign_operator_edge(&", ti, ", g_list_nth_data ( ", sg->code, "->list, ", tc, " ) );\n");
        node->code = strRightCatAlloc(node->code, "", 3,
            //INDENT[node->scope[0]], ti, " = g_list_nth_data ( ", sg->code, "->list, ", tc, " );\n",
            rt->code,
            INDENT[node->scope[0]], "} //END_OF_FOREACH\n");
        free(tlen);free(tc);
    }
    else {
        ERRNO = ErrorForeachType;
        errorInfo(ERRNO, node->line, "foreach has wrong type\n");
        return ERRNO;
    }
}

```

```

        break;
    }
}
/*****
case AST_JUMP_BREAK :
    // AUTHOR : Jing
    if(inLoop==0) {
        ERRNO = ErrorCallBreakOutsideOfLoop;
        errorInfo(ERRNO, node->line, "call 'break' outside of loop\n");
        return ERRNO;
    } else {
        char * freecode = NULL, * bkcode = NULL;
        // get all scope ids from the Loopbody to self
        int found = 0;
        GList * allscope = getAllScopeIdInside(LoopBody, NULL, node, &found);
        if (found == 0) {
            fprintf(stderr, "coding wrong for getAllScopeIdInside !!!!!\n");
        }
        // free code for GC
        int tl = g_list_length ( allscope );
        int i;
        for ( i=0; i<tl; i++ ) {
            int * pi = g_list_nth_data ( allscope, i );
            char * tcode = allFreeCodeInScope( *pi, NULL, node->scope[0] );
            freecode = strRightCatAlloc( freecode, "", 1, tcode );
            free(tcode);
        }
        g_list_free( allscope );
        // break code
        bkcode = strCatAlloc("", 2, INDENT[node->scope[0]], "break ;\n");
        // all
        node->code = strCatAlloc("", 2, freecode, bkcode);
        free(freecode);
        free(bkcode);
    }
    break;
case AST_JUMP_CONTINUE : {
    if(inLoop==0) {
        ERRNO = ErrorCallContinueOutsideOfLoop;
        errorInfo(ERRNO, node->line, "call 'continue' outside of loop\n");
        return ERRNO;
    } else {
        char * freecode = NULL, * ctcode = NULL;
        // get all scope ids from the Loopbody to self
        int found = 0;
        GList * allscope = getAllScopeIdInside(LoopBody, NULL, node, &found);
        if (found == 0) {
            fprintf(stderr, "coding wrong for getAllScopeIdInside !!!!!\n");
        }
        // free code for GC
        int tl = g_list_length ( allscope );
        int i;
        for ( i=0; i<tl; i++ ) {
            int * pi = g_list_nth_data ( allscope, i );
            char * tcode = allFreeCodeInScope( *pi, NULL, node->scope[0] );
            freecode = strRightCatAlloc( freecode, "", 1, tcode );
            free(tcode);
        }
        g_list_free( allscope );
        // continue code
        ctcode = strCatAlloc("", 4 , INDENT[node->scope[0]], "goto ", LoopGotoLabel, ";\n");
        node->code = strCatAlloc("", 2, freecode, ctcode);
        free(ctcode);
        free(freecode);
    }
    break;
}
case AST_JUMP_RETURN : {
    if(inFunc<0 && inFuncLiteral<0 ) {
        ERRNO = ErrorCallReturnOutsideOfFunc;
        errorInfo(ERRNO, node->line, "call 'return' outside of function or function literal\n");
        return ERRNO;
    }
    else {
        int rtype;
        if(isFunc == 1) { // FUNC
            rtype = * (int *) g_list_nth_data ( returnList, inFunc ); // obtain return type
            (* (int *) g_list_nth_data ( noReturn, inFunc ) ) ++ ; // count number of returns
        }
        else if (isFunc == -1) { // FL
            rtype = * (int *) g_list_nth_data ( returnList2, inFuncLiteral );
            (* (int *) g_list_nth_data ( noReturn2, inFuncLiteral ) ) ++;
        }
    }
}

```

```

    }
    else {
        fprintf(stderr, "wrong in return for isFunc\n");
        return -1;
    }
    char * freecode = NULL, * rtcode = NULL;
    int iptr = (rtype == VLIST_T || rtype == ELIST_T || rtype == VERTEX_T || rtype == EDGE_T || rtype ==
        GRAPH_T || rtype == STRING_T) ? 1 : 0;
    // type checking and return code
    if (node->nch == 0) {
        if (rtype != VOID_T) {
            ERRNO = ErrorInvalidReturnType;
            errorInfo(ERRNO, node->line, "invalid return type.\n");
            return ERRNO;
        }
        rtcode = strCatAlloc("", 2, INDENT[node->scope[0]], "return ;\n");
    }
    else {
        codeGen(node->child[0]);
        node->code = codeFrontDecl( node->scope[0]); // collect front code
        rtcode = codeFrontDecl(node->scope[0] );
        if (rtype != node->child[0]->type && node->child[0]->type >= 0) {
            ERRNO = ErrorInvalidReturnType;
            errorInfo(ERRNO, node->line, "invalid return type.\n");
            return ERRNO;
        }
        else if (rtype == node->child[0]->type && node->child[0]->type >= 0) {
            char * tmp = tmpReturnTmp();
            node->code = strRightCatAlloc(node->code, "", 7,
                INDENT[node->scope[0]], sTypeName(rtype), iptr ? " * ":" ", tmp, " = ",
                node->child[0]->code, ";\n");
            rtcode = strCatAlloc("", 4, INDENT[node->scope[0]], "return ", tmp, ";\n");
        }
        else {
            char * tmp = tmpReturnTmp();
            node->code = strRightCatAlloc(node->code, "", 7,
                INDENT[node->scope[0]], sTypeName(rtype), iptr ? " * ":" ", tmp, " = ",
                codeGetAttrVal( node->child[0]->code, rtype, node->line ),
                ";\n");
            rtcode = strCatAlloc("", 4, INDENT[node->scope[0]], "return ", tmp, ";\n");
        }
    }
    // get all scope ids from the funcbody to this return
    struct Node * gfb = NULL;
    if (isFunc == 1) gfb = FuncBody;
    else if (isFunc == -1) gfb = FuncLiteralBody;
    int found = 0;
    GList * allscope = getAllScopeIdInside(
        gfb, NULL, node, &found);
    if (found == 0) {
        fprintf(stderr, "coding wrong for getAllScopeIdInside at %d!!!!\n", node->line);
    }
    // freecode for GC
    int tl = g_list_length ( allscope );
    int i;
    for ( i=0; i<tl; i++ ) {
        int * pi = g_list_nth_data ( allscope, i );
        char * tcode = allFreeCodeInScope( *pi, FuncParaList, node->scope[0] );
        freecode = strRightCatAlloc( freecode, "", 1, tcode );
        free(tcode);
    }
    g_list_free( allscope );
    node->code = strRightCatAlloc( node->code, "", 2, freecode, rtcode );
}
break;
// 1> break continue is in scope of loop // DONE
// 2> return is in scope of func, and return type is correct //DONE
}

/*****
case AST_FUNC : { // AUTHOR : Jing
    lf = node->child[0]; // return type
    sg = node->child[1]; // parameter_list
    rt = node->child[2]; // compound_statement
    codeGen(sg);
    int zero = 0, nort;
    int oldisFunc = isFunc;
    inFunc++; isFunc = 1;
    // set para list to global
    if(sg->nch == 1) FuncParaList = NULL;
    else FuncParaList = getAllParaInFunc(sg->child[1], NULL);
*****/

```

```

// get all scope ids in the body of func
FuncBody = rt;
// get return type and number returns
returnList = g_list_append(returnList, (gpointer) &(lf->lexval.ival) );
noReturn = g_list_append( noReturn, (gpointer) &zero);
codeGen(rt);
returnList = g_list_remove(returnList, g_list_nth_data(returnList, inFunc) );
gpointer gp = g_list_nth_data( noReturn, inFunc );
nort = *(int *) gp;
noReturn = g_list_remove( noReturn, gp );
// clean lists
char * initcode = allInitTmpVabCodeInScope( rt->scope[1], FuncParaList, 1 );
g_list_free( FuncParaList );
FuncBody = NULL;
inFunc--; isFunc = oldisFunc;
if ( nort <= 0 ) {
    ERRNO = ErrorNoReturnInFunc;
    errorInfo(ERRNO, node->line, "missing return in function declaration.\n");
    free(initcode);
    return ERRNO;
}
int flag0 = 0;
int type0 = lf->lexval.ival;
if (type0 == VERTEX_T || type0 == EDGE_T || type0 == VLIST_T ||
    type0 == ELIST_T || type0 == STRING_T || type0 == GRAPH_T) flag0 = 1;
node->code = strCatAlloc("", 10,
    sTypeName(lf->lexval.ival), (flag0)? " * ":" ",
    node->symbol->bind, // func_id
    " ( ", sg->code, " ) ", "{\n",
    initcode,
    rt->code, "}\n");
node->codetmp = strCatAlloc("", 6,
    sTypeName(lf->lexval.ival), (flag0)? " * ":" ",
    node->symbol->bind, // func_id
    " ( ", sg->code, " );\n " );
free(initcode);
//showASTandST(node, "Function Definition");
break;
}
case FUNC_LITERAL : { // function_literal_declaration
    lf = node->child[1]; // return type
    sg = node->child[0]; // parameter_list
    rt = node->child[2]; // compound_statement
    codeGen(sg);
    int zero = 0, nort;
    if (inFuncLiteral >= 0 ) {
        ERRNO = ErrorNestedFuncLiteralInFuncLiteral;
        errorInfo(ERRNO, node->line, "nested function literal in another function literal.\n");
        return ERRNO;
    }
    int oldisFunc = isFunc;
    inFuncLiteral++; isFunc = -1;
    // set para list to global
    if (sg->nch == 1) FuncParaList = NULL;
    else FuncParaList = getAllParaInFunc(sg->child[1], NULL);
    // set body pointer to global (used in AST_RETURN)
    FuncLiteralBody = rt;
    // get return type and number returns
    returnList2 = g_list_append(returnList2, (gpointer) &(lf->lexval.ival) );
    noReturn2 = g_list_append( noReturn2, (gpointer) &zero);
    codeGen(rt);
    returnList2 = g_list_remove(returnList2, g_list_nth_data(returnList2, inFuncLiteral) );
    gpointer gp = g_list_nth_data( noReturn2, inFuncLiteral );
    nort = *(int *) gp;
    noReturn2 = g_list_remove( noReturn2, gp );
    // clean lists
    char * initcode = allInitTmpVabCodeInScope( rt->scope[1], FuncParaList, 1 );
    g_list_free( FuncParaList );
    FuncLiteralBody = NULL;
    inFuncLiteral--; isFunc = oldisFunc;
    if ( nort <= 0 ) {
        ERRNO = ErrorNoReturnInFunc;
        errorInfo(ERRNO, node->line, "missing return in function literal declaration.\n");
        free(initcode);
        return ERRNO;
    }
    int flag0 = 0;
    int type0 = lf->lexval.ival;
    if (type0 == VERTEX_T || type0 == EDGE_T || type0 == VLIST_T ||
        type0 == ELIST_T || type0 == STRING_T || type0 == GRAPH_T) flag0 = 1;
    // put code to node->codetmp, as we need put all func_literals in

```

```

// the external in target code.
node->code = strCatAlloc(" ", 11,
    sTypeName(lf->lexval.ival), (flag0)? " * " : " ",
    node->symbol->bind, // func_id
    " ( void * _obj, int _obj_type",
    (sg->nch==1)? " " : " ",
    sg->code, " ) ", "{\n",
    initcode,
    rt->code, " } // END_OF_FUNC_LITERAL\n");
node->codetmp = strCatAlloc(" ", 7,
    sTypeName(lf->lexval.ival), (flag0)? " * " : " ",
    node->symbol->bind, // func_id
    " ( void * _obj, int _obj_type",
    (sg->nch==1)? " " : " ",
    sg->code, " );\n");
free(initcode);
break;
}
/*****
case AST_FUNC_DECLARATOR : // AUTHOR : Jing
// here only create parameter list
if (node->nch==1) // empty list
    node->code = strCatAlloc(" ", 1, "");
else {
    sg = node->child[1];
    codeGen(sg);
    node->code = strCatAlloc(" ", 1, sg->code);
}
break;
case AST_PARA_DECLARATION :
    lf = node->child[0]; // declaration_specifiers
    rt = node->child[1]; // IDENTIFIER or attribute
    codeGen(rt);
    node->type = node->child[0]->lexval.ival;
    if (node->type == STRING_T || node->type == VLIST_T || node->type == ELIST_T ||
        node->type == VERTEX_T || node->type == EDGE_T || node->type == GRAPH_T)
        node->code = strCatAlloc(" ", 3, sTypeName(lf->lexval.ival), " * ", node->child[1]->code);
    else if (node->type == BOOL_T || node->type == INT_T || node->type == FLOAT_T)
        node->code = strCatAlloc(" ", 3, sTypeName(lf->lexval.ival), " ", node->child[1]->code);
    else {
        ERRNO = ErrorWrongArgumentType;
        errorInfo(ERRNO, node->line, "invalid argument type.\n");
        return ERRNO;
    }
    break;
/*****
case AST_PRINT_STAT : // AUTHOR : Chantal, Kunal
    codeGen(node->child[0]);
    node->code = codeFrontDecl(node->scope[0]);
    node->code = strRightCatAlloc(node->code, " ", 1, node->child[0]->code);
    break;
case AST_PRINT_COMMA :
    codeGen(node->child[0]);
    codeGen(node->child[1]);
    node->code = strCatAlloc(" ", 2, node->child[0]->code, node->child[1]->code);
    break;
case AST_PRINT : {
    codeGen(node->child[0]);
    int tt = node->child[0]->type;
    if ( tt == BOOL_T || tt == INT_T || tt == FLOAT_T ) {
        node->code = strCatAlloc(" ", 6,
            INDENT[node->scope[0]], "print_", typeMacro(tt), " ( ",
            node->child[0]->code, " );\n");
    }
    else if ( tt == VLIST_T || tt == ELIST_T || tt == VERTEX_T ||
        tt == EDGE_T || tt == GRAPH_T || tt == STRING_T || tt == DYN_ATTR_T ) {
        SymbolTableEntry* e = tmpVab( tt, node->scope[1] );
        char * cass = tmpVabAssign( e, node->child[0]->code );
        frontDeclExp = strRightCatAlloc( frontDeclExp, " ", 1, cass);
        if (tt != DYN_ATTR_T)
            node->code = strCatAlloc ( " ", 6,
                INDENT[node->scope[0]], "print_", typeMacro(tt), " ( ",
                e->bind, " );\n");
        else
            node->code = strCatAlloc(" ", 4,
                INDENT[node->scope[0]], "print_attr ( ",
                e->bind, " );\n");
    }
}
else {
    ERRNO = ErrorPrintWrongType;
    errorInfo( ERRNO, node->line, "print wrong type.\n");
}

```

```

        return ERRNO;
    }
    break;
}
/*****/
case AST_READ_GRAPH: // FILE >> Graph // AUTHOR : Chantal, Kunal
    lf=node->child[0];
    rt=node->child[1];
    codeGen(lf); codeGen(rt);
    if ( lf->type == STRING_T && rt->type == GRAPH_T ) {
        char * tg = tmpGraphVab();
        node->code = strCatAlloc("", 7,
            INDENT[node->scope[0]], assignFunc(GRAPH_T), " ( &( ",
                rt->code, " ) , readGraph( ", lf->code, "->str );\n"
            );
    }
    else {
        ERRNO = ErrorTypeMismatch;
        errorInfo(ERRNO, node->line, "expected 'FILE:STRING' to be fetched from 'GRAPH' file location.\n" );
    }
    break;
case AST_WRITE_GRAPH: // FILE << Graph
    lf=node->child[0];
    rt=node->child[1];
    codeGen(lf); codeGen(rt);
    if ( lf->type == STRING_T && rt->type == GRAPH_T ) {
        node->code = strCatAlloc("", 6,
            INDENT[node->scope[0]], "saveGraph( ", rt->code, " , (", lf->code,")->str );\n"
        );
    }
    else {
        ERRNO = ErrorTypeMismatch;
        errorInfo(ERRNO, node->line, "expected 'FILE:STRING' to be written into 'GRAPH' file location.\n");
    }
    break;
/*****/
default:
    if(node->code == NULL) {
#ifdef _DEBUG
        debugInfo("Warning: No code generated on ");
        astOutNode(node, DEBUGIO," \n");
#endif
    }
    return 0;
}

// AUTHOR : Lixing
void derivedTypeInitCode(struct Node* node, int type, int isglobal){
    if(node->token == AST_COMMA){
        derivedTypeInitCode(node->child[0], type, isglobal);
        derivedTypeInitCode(node->child[1], type, isglobal);
        node->code = strCatAlloc("",2, node->child[0]->code, node->child[1]->code);
        if (node->scope[0]==0) node->codetmp = strCatAlloc("",3,node->child[0]->codetmp,"",node->child[1]->codetmp);
    }else if (node->token == IDENTIFIER) {
        codeGen(node);
        node->code = strCatAlloc("",3 ,INDENT[node->scope[0]],node->symbol->bind," = NULL; ");
        switch(type){
            case GRAPH_T:
                node->code = strRightCatAlloc(node->code,"",3 ,"assign_operator_graph ( &( ",
                    node->symbol->bind," ) , new_graph() );\n");
                break;
            case VERTEX_T:
                node->code = strRightCatAlloc(node->code,"",3 ,"assign_operator_vertex ( &( ",
                    node->symbol->bind," ) ,new_vertex() );\n");
                break;
            case EDGE_T:
                node->code = strRightCatAlloc(node->code,"",3 ,"assign_operator_edge ( &( ",
                    node->symbol->bind," ) ,new_edge() );\n");
                break;
            default:
                break;
        }
    }
    else if (node->token == AST_ASSIGN) {
        codeGen(node->child[0]); codeGen(node->child[1]);
        if (node->child[1]->type != type) {
            ERRNO= ErrorInitDerivedType;
            errorInfo(ERRNO, node->line, "type mismatch for the initialization of derived type.\n");
        }
    }
}

```

```

        node->code = NULL;
        return;
    }
    node->code = strCatAlloc("", 3, INDENT[node->scope[0]], node->child[0]->symbol->bind, " = NULL; ");
    switch(type) {
        case GRAPH_T:
            node->code = strRightCatAlloc(node->code, "", 5, "assign_operator_graph ( &( ",
                node->child[0]->symbol->bind, " ) , ", node->child[1]->code, " );\n");
            break;
        case VERTEX_T:
            node->code = strRightCatAlloc(node->code, "", 5, "assign_operator_vertex ( &( ",
                node->child[0]->symbol->bind, " ) , ", node->child[1]->code, " );\n");
            break;
        case EDGE_T:
            node->code = strRightCatAlloc(node->code, "", 5, "assign_operator_edge ( &( ",
                node->child[0]->symbol->bind, " ) , ", node->child[1]->code, " );\n");
            break;
        default:
            break;
    }
    if(node->scope[0]==0) node->codetmp = strCatAlloc("", 1, node->child[0]->codetmp);
}
else {
    ERRNO = ErrorIllegalDerivedTypeDeclaration;
    errorInfo(ERRNO, node->line, "Illegal declaration of derived type (vertex, edge, graph).\n");
}
}

// AUTHOR : Lixing
void stringInitCode(struct Node* node, int type, int isglobal){
    if(node->token == AST_COMMA){
        stringInitCode(node->child[0], type, isglobal);
        stringInitCode(node->child[1], type, isglobal);
        node->code = strCatAlloc("", 2, node->child[0]->code, node->child[1]->code);
        if(node->scope[0]==0) node->codetmp = strCatAlloc("", 3, node->child[0]->codetmp, ",", node->child[1]->codetmp);
    } else if (node->token == AST_ASSIGN){
        codeGen(node->child[0]); codeGen(node->child[1]);
        //if(isglobal)
            node->code = strCatAlloc("", 6, INDENT[node->scope[0]], "assign_operator_string( &( ",
                node->child[0]->symbol->bind, " ) , ", node->child[1]->code, " );\n");
        //else
            node->code = strCatAlloc("", 7, INDENT[node->scope[0]],
                sTypeName(type), " * ", node->child[0]->symbol->bind, " = ", node->child[1]->code, " );\n");
        if(node->scope[0]==0) node->codetmp = strCatAlloc("", 1, node->child[0]->codetmp);
    } else {
        codeGen(node);
        //if(isglobal)
            node->code = strCatAlloc("", 4, INDENT[node->scope[0]], "assign_operator_string( &( ",
                node->symbol->bind, " ) , g_string_new(\"\\\") );\n");
        //else
            node->code = strCatAlloc("", 5, INDENT[node->scope[0]], sTypeName(type), " * ", node->symbol->bind, " =
                g_string_new(\"\\\") );\n");
    }
}

// AUTHOR : Lixing
void listInitCode(struct Node* node, int type, int isglobal){
    int mtype = (type == VLIST_T) ? VERTEX_T : EDGE_T;
    if(node->token == AST_COMMA){
        listInitCode(node->child[0], type, isglobal);
        listInitCode(node->child[1], type, isglobal);
        node->code = strCatAlloc("", 2, node->child[0]->code, node->child[1]->code);
        if(node->scope[0]==0) node->codetmp = strCatAlloc("", 3, node->child[0]->codetmp, ",", node->child[1]->codetmp);
    }
    else if (node->token == AST_ASSIGN){
        codeGen(node->child[0]); codeGen(node->child[1]);
        char num[32];
        int flag = listCountCheck(node->child[1], mtype);
        int nArgs = (flag > 0)? flag : 0;
        sprintf(num, "%d\0", nArgs);
        node->code = strCatAlloc("", 9, INDENT[node->scope[0]],
            (isglobal)? "" : "ListType * ",
            "", node->child[0]->symbol->bind,
            " = NULL; assign_operator_list ( &( ", node->child[0]->symbol->bind,
            " ) , list_declaration( ", typeMacro(mtype), " , ", num);
        if(nArgs>0) node->code = strRightCatAlloc( node->code, "", 3, " , ", node->child[1]->code, " );\n");
        else node->code = strRightCatAlloc( node->code, "", 1, " );\n");
        // if not init by [],
        if (flag<0) {
            char * fc = codeFrontDecl( node->scope[0] );
            node->code = strRightCatAlloc( node->code, "", 6,

```



```

        fc,
        "assign_operator_list ( & ( ", node->child[0]->symbol->bind, " ) , ( ",
        node->child[1]->code, " ) );\n";
    free(fc);
}
if(node->scope[0]==0) node->codetmp = strCatAlloc("",1,node->child[0]->codetmp);
}
else { // empty list
    codeGen(node);
    node->code = strCatAlloc("", 8, INDENT[node->scope[0]],
//      (isglobal)? "" : "ListType * ",
        "",node->symbol->bind,
        " = NULL; assign_operator_list ( & ( ", node->symbol->bind, " ) , list_declaration( ", typeMacro(mtype),
        " , 0 ) );\n");
}
}

// AUTHOR : Jing
// count number of initializor in [ ... ]
int listCountCheck(struct Node* node, int type){
    int count = 0, flag = 0;
    struct Node* tn = node;
    if(tn->token != AST_LIST_INIT) {
        return -1;
    }
    if(tn->nch > 0) {
        tn = tn->child[0];
        while (tn->token == AST_COMMA ) {
            if (tn->child[1]->token != IDENTIFIER) { flag = ErrorAssignmentExpression; break; }
            if ( tn->child[1]->type != type ) { flag = ErrorListMixedType; break; }
            tn = tn->child[0];
            count++;
        }
        if (tn->token == IDENTIFIER && flag == 0) {
            if ( tn->type != type ) flag = ErrorListMixedType;
            count++;
        }
        //} // disable assignment in [ ... ]
        //else if(tn->token == AST_ASSIGN){
        //    if ( tn->type != type ) flag = ErrorListMixedType;
        //    count++;
        //}else{
        flag = ErrorAssignmentExpression;
        //}
        if (flag == ErrorListMixedType) {
            ERRNO = flag;
            errorInfo(ERRNO, node->line, "list Initialization with wrong type.\n");
        }
        else if(flag == ErrorAssignmentExpression){
            ERRNO = flag;
            errorInfo(ERRNO, node->line, "list Initialization with wrong argument expression.\n");
        }
    }
    return count;
}

// AUTHOR : Jing
int codeAttr ( struct Node * node ) {
    // codeGen should already be called on this node, before codeAttr
    char * code = node->code;
    if(node->type<=0 || node->type>STRING_T) {
        ERRNO = ErrorBinaryOperationWithDynamicType;
        errorInfo(ERRNO, node->line, "Binary Operation with Dynamic Type.\n");
        return 1;
    }
    SymbolTableEntry* e = tmpVab( DYN_ATTR_T, node->scope[1] );
    frontDeclExp = strRightCatAlloc( frontDeclExp, "",8 ,
        INDENT[node->scope[0]], "assign_operator_attr( & ( ",
        e->bind, " ) , new_attr_", typeMacro(node->type),
        " ( ", code," ) );\n");
    node->code = strCatAlloc("", 1, e->bind);
    free(code);
    return 0;
}

// AUTHOR : Jing
char * codeGetAttrVal( char * operand, int type, int lno ) {
    if(type != BOOL_T && type != INT_T && type != FLOAT_T && type != STRING_T) {
        ERRNO = ErrorGetAttrForWrongType;
        errorInfo(ERRNO, lno, "get attribute value for wrong type.\n");
        return NULL;
    }
}

```

```

    return strCatAlloc("", 7, "get_attr_value_", typeMacro(type),
        " ( ", operand, " , ", strLine(lno), " ) " );
}

// AUTHOR : Jing
char * codeFrontDecl(int lvl ) {
    char * decl = NULL;
    if(1||existMATCH == 1 || existPIPE == 1){ // for MATCH
        decl = strRightCatAlloc(decl, " ", 2, INDENT[lvl], frontDeclExp);
        free(frontDeclExp); frontDeclExp = NULL;
        existMATCH = 0; existPIPE = 0;
    }
    return decl;
}

// AUTHOR : Jing
int codeAssignLeft( struct Node * node) {
    if (node->token == IDENTIFIER) {
        codeGen(node);
    }
    else if (node->token == AST_ATTRIBUTE) {
        // assume: NO assignment in MATCH
        node->child[0]->code = strCatAlloc("", 1, node->child[0]->symbol->bind);
        node->child[1]->code = strCatAlloc("", 1, node->child[1]->lexval.sval);
        SymbolTableEntry* et = tmpVab( DYN_ATTR_T, node->scope[1] );
        char * code = NULL;
        // put "1" for xxx_get_attribute to auto allocate storage
        if(node->child[0]->type == VERTEX_T )
            code = strCatAlloc("", 7, "vertex_get_attribute( ",
                node->child[0]->code, " , \\"", node->child[1]->code, "\"", 1, " , strLine(node->line), " )");
        else if(node->child[0]->type == EDGE_T )
            code = strCatAlloc("", 7, "edge_get_attribute( ",
                node->child[0]->code, " , \\"", node->child[1]->code, "\"", 1, " , strLine(node->line), " )");
        else {
            ERRNO = ErrorGetAttrForWrongType;
            errorInfo(ERRNO, node->line, "Access attribute for type '%s'.\n",
                sTypeName(node->child[0]->type) );
        }
        char * cass = tmpVabAssign( et, code );
        frontDeclExp = strRightCatAlloc( frontDeclExp, " ", 1, cass );
        node->code = strCatAlloc( " ", 1, et->bind );
        free(code); free(cass);
        node->type = DYNAMIC_T;
    }
    else if (node->token == DYN_ATTRIBUTE) {
        SymbolTableEntry* et = tmpVab( DYN_ATTR_T, node->scope[1] );
        char* code = strCatAlloc("", 6,
            "object_get_attribute( _obj, _obj_type, ",
            "\":", node->lexval.sval, "\"", 1, " , strLine(node->line), " ) ");
        char * cass = tmpVabAssign( et, code );
        frontDeclExp = strRightCatAlloc( frontDeclExp, " ", 1, cass );
        node->code = strCatAlloc( " ", 1, et->bind );
        free(code); free(cass);
        node->type = DYNAMIC_T;
    }
}

// AUTHOR : Jing
int codeFuncWrapDynArgs(struct Node* node, GArray* tcon, int* cnt){
    if(node->token == AST_COMMA) {
        codeFuncWrapDynArgs(node->child[0], tcon, cnt);
        codeFuncWrapDynArgs(node->child[1], tcon, cnt);
        node->code = strCatAlloc("", 3, node->child[0]->code, " , ", node->child[1]->code);
    }
    else if (node->token == AST_ARG_EXPS) {
        codeGen(node);
        if(tcon->len > *cnt) {
            int rtype = g_array_index(tcon, int, *cnt);
            if(node->type < 0) {
                char * ctmp = node->code;
                node->code = codeGetAttrVal(ctmp, rtype, node->line);
                free(ctmp);
            }
            else if (node->type >= 0 && node->type != rtype ) {
                ERRNO = ErrorFunctionCallIncompatibleParameterType;
                errorInfo(ERRNO, node->line, "function arg has incompatible arguments to its declaration.\n");
            }
        }
        (*cnt)++;
    }
    return 0;
}

```

```

}

// AUTHOR : Jing
char * codeForFreeDerivedVabInScope(ScopeId sid, int type, GList * gl, ScopeId lvl, int which){
    GList * vals = NULL;
    if (which == 0) vals = sTableAllVarScope( sid, type );
    else if (which == 1) vals = tmpTableAllVarScope( sid, type );

    char * code = NULL, * freefunc = codeFreeFuncName(type);
    int i, l = g_list_length( vals );
    SymbolTableEntry * e;
#ifdef _DEBUG
    int ll = g_list_length( gl );
    debugInfo("codeForFreeDerivedVabInScope: sid=%d, type=%d, l=%d, ll=%d\n", sid,type,l,ll);
    if(ll>0) {
        int i;
        for (i=0; i<ll; ++i) {
            e = (SymbolTableEntry *) g_list_nth_data( gl, i );
            debugInfoExt("    gl[%d] ==> %s\n", i, e->bind);
        }
    }
#endif
    for ( i=0; i<l; ++i ){
        e = (SymbolTableEntry *) g_list_nth_data( vals, i );
        if( g_list_find( gl, (gpointer) e ) == NULL )
            code = strRightCatAlloc( code, "", 7, INDENT[lvl], freefunc, "( ", e->bind, " );", e->bind, " = NULL;\n" );
    }
    g_list_free( vals );
    return code;
}

// AUTHOR : Jing
char * codeForInitTmpVabInScope ( ScopeId sid, int type, GList * gl, ScopeId lvl, int which ){
    GList * vals = NULL;
    if (which == 0) vals = sTableAllVarScope( sid, type );
    else if (which == 1) vals = tmpTableAllVarScope( sid, type );

    char * code = NULL, * freefunc = codeFreeFuncName(type);
    int i, l = g_list_length( vals );
    SymbolTableEntry * e;
    int isptr = ( type == VLIST_T || type == ELIST_T || type == GRAPH_T || type == VERTEX_T ||
        type == EDGE_T || type == DYN_ATTR_T || type == STRING_T ) ? 1: 0 ;
    char * def;
    switch (type) {
        case BOOL_T: def = "false"; break;
        case INT_T: def = "0"; break;
        case FLOAT_T: def = "0.0"; break;
        default: def = "NULL"; break;
    }
    for ( i=0; i<l; ++i ){
        e = (SymbolTableEntry *) g_list_nth_data( vals, i );
        if( g_list_find( gl, (gpointer) e ) == NULL ) {
            if(sid!=0 || which == 1)
                code = strRightCatAlloc( code, "", 7, INDENT[lvl], sTypeName(e->type),
                    (isptr) ? " * ": " ", e->bind, " = ", def, ";\n");
            else
                code = strRightCatAlloc( code,"", 5, INDENT[lvl],
                    e->bind, " = ", def, ";\n");
        }
    }
    g_list_free( vals );
    return code;
}

// AUTHOR : Jing
char * allFreeCodeInScope(ScopeId sid, GList * gl, ScopeId lvl) {
    char * sc = codeForFreeDerivedVabInScope( sid, STRING_T, gl, lvl, 0 );
    char * vc = codeForFreeDerivedVabInScope( sid, VERTEX_T, gl, lvl, 0 );
    char * ec = codeForFreeDerivedVabInScope( sid, EDGE_T, gl, lvl, 0 );
    char * gc = codeForFreeDerivedVabInScope( sid, GRAPH_T, gl, lvl, 0 );
    char * vlc = codeForFreeDerivedVabInScope( sid, VLIST_T, gl, lvl, 0 );
    char * elc = codeForFreeDerivedVabInScope( sid, ELIST_T, gl, lvl, 0 );
    char * tsc = codeForFreeDerivedVabInScope( sid, STRING_T, gl, lvl, 1 );
    char * tvl = codeForFreeDerivedVabInScope( sid, VERTEX_T, gl, lvl, 1 );
    char * tec = codeForFreeDerivedVabInScope( sid, EDGE_T, gl, lvl, 1 );
    char * tgc = codeForFreeDerivedVabInScope( sid, GRAPH_T, gl, lvl, 1 );
    char * tvlc = codeForFreeDerivedVabInScope( sid, VLIST_T, gl, lvl, 1 );
    char * telc = codeForFreeDerivedVabInScope( sid, ELIST_T, gl, lvl, 1 );
    char * tatt = codeForFreeDerivedVabInScope( sid, DYN_ATTR_T, gl, lvl, 1 );

    char * rlt = strCatAlloc("", 13,

```

```

        tatt, sc, tsc, vlc, elc, tvlc, telc,
        ec, tec, vc, tvc, gc, tgc);
free(sc);free(vc);free(ec);free(gc);free(vlc);free(elc);
free(tsc);free(tvc);free(tec);free(tgc);free(tvlc);free(telc);free(tatt);
return rlt;
}

// AUTHOR : Jing
char * allInitTmpVabCodeInScope(ScopeId sid, GList * gl, ScopeId lvl) {
    char * sc = codeForInitTmpVabInScope( sid, STRING_T, gl, lvl, 0 );
    char * vc = codeForInitTmpVabInScope( sid, VERTEX_T, gl, lvl, 0 );
    char * ec = codeForInitTmpVabInScope( sid, EDGE_T, gl, lvl, 0 );
    char * gc = codeForInitTmpVabInScope( sid, GRAPH_T, gl, lvl, 0 );
    char * vlc = codeForInitTmpVabInScope( sid, VLIST_T, gl, lvl, 0 );
    char * elc = codeForInitTmpVabInScope( sid, ELIST_T, gl, lvl, 0 );

    char * tsc = codeForInitTmpVabInScope( sid, STRING_T, gl, lvl, 1 );
    char * tvc = codeForInitTmpVabInScope( sid, VERTEX_T, gl, lvl, 1 );
    char * tec = codeForInitTmpVabInScope( sid, EDGE_T, gl, lvl, 1 );
    char * tgc = codeForInitTmpVabInScope( sid, GRAPH_T, gl, lvl, 1 );
    char * tvlc = codeForInitTmpVabInScope( sid, VLIST_T, gl, lvl, 1 );
    char * telc = codeForInitTmpVabInScope( sid, ELIST_T, gl, lvl, 1 );
    char * tatt = codeForInitTmpVabInScope( sid, DYN_ATTR_T, gl, lvl, 1 );

    char * rlt = strCatAlloc(" ", 13,
        tatt, sc, tsc, vlc, elc, tvlc, telc,
        ec, tec, vc, tvc, gc, tgc);

    free(sc);free(vc);free(ec);free(gc);free(vlc);free(elc);
    free(tsc);free(tvc);free(tec);free(tgc);free(tvlc);free(telc);free(tatt);
    return rlt;
}

// AUTHOR : Jing
GList * getAllParaInFunc(struct Node * node, GList * gl) {
    if (node == NULL) return gl;
    else if (node->token == AST_COMMA) {
        gl = getAllParaInFunc(node->child[0], gl);
        gl = getAllParaInFunc(node->child[1], gl);
    }
    else if (node->token == AST_PARA_DECLARATION) {
        gl = g_list_append( gl, node->child[1]->symbol );
    }
    else {
        fprintf(stderr, "getAllParaInFunc: unknow node %d !!!!!!!!!\n", node->token);
    }
    return gl;
}

// AUTHOR : Jing
GList * getReturnVab( struct Node * node, GList * gl) {
    if (node == NULL) return gl;
    else if (node->token == AST_JUMP_RETURN) {
        if (node->nch!=0) {
            gl = g_list_append( gl, node->child[0]->symbol );
        }
        return gl;
    }

    int i;
    for (i=0; i<node->nch; ++i) {
        gl = getReturnVab( node->child[i], gl );
    }
    return gl;
}

// AUTHOR : Jing
GList * getAllScopeIdInside( struct Node * node, GList * gl, struct Node * target, int * rlt) {
    if (node == NULL) return gl;
    int flag = (node == target);
    if (flag == 0) { // I am not target, try my child
        int i;
        for (i=0; i<node->nch; ++i) {
            gl = getAllScopeIdInside( node->child[i], gl, target, &flag );
            if(flag != 0) break; // only one path
        }
    }
    if (flag == 1) { // find it
        int tl = g_list_length( gl );
        int i, fflag = 0;
        for ( i=0; i<tl; i++ ) { // check duplicate

```

```

        int * ii = g_list_nth_data( gl , i );
        if ( *ii == node->scope[1] ) { fflag = 1; break; }
    }
    if (!fflag) gl = g_list_append( gl, &(node->scope[1]) );
}
*rlt = flag;
return gl;
}

// AUTHOR : Jing
int codeAllGen(struct Node* node, char ** mainCode, char ** funCode) {
    if(node->token == AST_EXT_STAT_COMMA) {
        codeAllGen(node->child[0], mainCode, funCode);
        codeAllGen(node->child[1], mainCode, funCode);
    }
    else if(node->token == AST_FUNC) { // merge in funCode
        codeGen(node);
        *funCode = strRightCatAlloc( *funCode, "", 2, node->code, "\n" );
    }
    else { // merge in mainCode
        codeGen(node);
        *mainCode = strRightCatAlloc( *mainCode, "", 1, node->code );
    }

    return 0;
}

// AUTHOR : Jing
void codeAllFuncLiteral(struct Node* node, char ** code) {
    // travel the entire tree, get all func_literals
    if (node==NULL) return;
    if (node->token == FUNC_LITERAL ) {
        *code = strRightCatAlloc( *code, "", 2, node->code, "\n" );
    }
    else if ( node->token == AST_MATCH ) {
        *code = strRightCatAlloc( *code, "", 2, node->codetmp, "\n" );
        // DO NOT return, for nested Func_Literal
        //if (node->token == FUNC_LITERAL) return;
    }
    int i;
    for (i=0; i<node->nch; ++i) {
        codeAllFuncLiteral(node->child[i], code);
    }
    return;
}

// AUTHOR : Jing
void codeInclude(char ** code) {
    *code = strRightCatAlloc( *code, "", 1,
        "#include \"nsbl.h\"\n" );
}

// AUTHOR : Jing
void codeAllGlobal(struct Node* node, char ** code) {
    // travel the entire tree, get all declaration in scope level 0
    if (node==NULL) return;
    if (node->token == AST_DECLARATION) {
        if (node->scope[0] == 0)
            *code = strRightCatAlloc( *code, "", 1, node->codetmp );
        return;
    }
    else if(node->token == AST_FUNC) {
        *code = strRightCatAlloc( *code, "", 1, node->codetmp );
    }
    else if (node->token == FUNC_LITERAL ) {
        *code = strRightCatAlloc( *code, "", 1, node->codetmp );
    }
    int i;
    for (i=0; i<node->nch; ++i) {
        codeAllGlobal(node->child[i], code);
    }
    return;
}

// AUTHOR : Jing
char * wapperMainCode(char * mainBodyCode){
    char * head = "int main() {\n\n";
    char * GC1 = "gcInit();\n";
    char * initcode = allInitTmpVabCodeInScope( 0, NULL, 0 );
    char * freecode = allFreeCodeInScope( 0, NULL, 0 );
#ifdef _DEBUG

```

```

        //debugInfo("MainFreeCode:\n");
        //debugInfoExt("%s",freecode);
#endif
    char * GC2 = "gcDel();\n";
    char * end = "\n} // END_OF_MAIN \n";
    return strCatAlloc(" ",7,head,GC1,initcode, mainBodyCode, freecode, GC2, end);
}

// AUTHOR : Jing
void exportCode(char * code){
    fprintf(OUTFILESTREAM,"%s",code);
}

```

../src/CodeGen.c

```

// author : Jing
#ifndef CODEGENUTIL_H_NSBL_
#define CODEGENUTIL_H_NSBL_
#include "SymbolTable.h"

// string operation
char * strCatAllocBase(char* sep, int n, char ** ptr);
char * strCatAlloc(char* sep, int n, ...);
char * strRightCatAlloc(char * base, char* sep, int n, ...);
void strFreeAll(int n, ...);
#define strCatAllocSpace(n,...)      strCatAlloc(' ',n,...)

// auxiliary funcs
char * strLine(int l);
char * codeFreeFuncName( int type );
char * codeRemoveAttrFuncName( int type );
char * opMacro(int ma);
char * DynOP(int ma);
char * typeMacro(int t);
char * assignFunc(int t);
char * gotolabel();
char * tmpReturnTmp();
char * tmpMatch();
char * tmpMatchStr();
char * tmpMatchStrVab();
char * tmpGraphVab();
SymbolTableEntry* tmpVab(int type, ScopeId sid);
char * tmpVabAssign( SymbolTableEntry* e, char * value );
char * tmpVabDel( SymbolTableEntry* e );
void codeIndentInit();
void codeIndentFree();

#endif

```

../src/CodeGenUtil.h

```

// author : Jing
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include "CodeGenUtil.h"
#include "Parser.tab.h"
#include "type.h"
#include "operator.h"
#include "global.h"

char ** INDENT;                // space indent

char * strCatAllocBase(char* sep, int n, char ** ptr){
    if(n<=0) return NULL;
    int i;
    int l, ll=0, ls=strlen(sep);
    for(i=0; i<n; ++i) ll += strlen(ptr[i]);

    char * nstr = (char *) malloc (sizeof(char)*(ll+(n-1)*ls+1));
    char * pt = nstr;
    for(i=0; i<n; ++i) {
        l = strlen(ptr[i]);
        strncpy(pt, ptr[i], l); pt += l;
        if(i<n-1) {
            strncpy(pt, sep, ls); pt += ls;
        }
        else *(pt++) = '\0';
    }
    return nstr;
}

```

```

}

char * strCatAlloc(char* sep, int n, ...) {
    if (n<=0) return NULL;
    int i, nsep=0;
    va_list args;
    va_start (args, n);
    char ** ptr = (char **) malloc (sizeof(char *)*n);
    for(i=0; i<n; ++i) {
        char *tp = va_arg(args, char *);
        if(tp!=NULL) ptr[nsep++] = tp;
    }
    char* nstr = strCatAllocBase(sep, nsep, ptr);
    free(ptr);
    va_end(args);
    return nstr;
}

char * strRightCatAlloc(char * base, char* sep, int n, ...) {
    if (n<=0) return base;
    int i, nsep=0;
    va_list args;
    va_start (args, n);
    char ** ptr = (char **) malloc (sizeof(char *)*(n+1));
    if(base!=NULL) ptr[nsep++] = base;
    for(i=0; i<n; ++i) {
        char *tp = va_arg(args, char *);
        if(tp!=NULL) ptr[nsep++] = tp;
    }
    char* nstr = strCatAllocBase(sep, nsep, ptr);
    if(base!=NULL) free(base);
    free(ptr);
    va_end(args);
    return nstr;
}

void strFreeAll(int n, ...) {
    if (n<=0) return;
    int i;
    va_list args;
    va_start (args, n);
    char * tp;
    for(i=0; i<n; ++i) {
        if ( (tp = va_arg(args, char *)) != NULL)
            free(tp);
    }
    return;
}

char * strLine(int l) {
    static char LineNO[64];
    sprintf(LineNO, "%d\0", l);
    return LineNO;
}

char * codeFreeFuncName( int type ) {
    switch (type) {
        case VLIST_T :
        case ELIST_T :
            return "destroy_list" ;
        case VERTEX_T :
            return "destroy_vertex" ;
        case EDGE_T :
            return "destroy_edge" ;
        case GRAPH_T :
            return "destroy_graph" ;
        case STRING_T :
            return "destroy_string" ;
        case DYN_ATTR_T :
            return "destroy_attr";
        default :
            return NULL;
    }
}

char * codeRemoveAttrFuncName( int type ) {
    switch ( type ) {
        case VERTEX_T :
            return "vertex_remove_attribute";
        case EDGE_T :
            return "edge_remove_attribute";
    }
}

```

```

        case GRAPH_T :
            return "graph_remove_attribute";
        default :
            return NULL;
    }
}

char * opMacro(int ma) {
    switch(ma) {
        case AST_ASSIGN :      return "=";
        case ADD_ASSIGN :      return "+=";
        case SUB_ASSIGN :      return "-=";
        case MUL_ASSIGN :      return "*=";
        case DIV_ASSIGN :      return "/=";
        case OR :              return "||";
        case AND :              return "&&";
        case EQ :              return "==";
        case NE :              return "!=";
        case LT :              return "<";
        case GT :              return ">";
        case LE :              return "<=";
        case GE :              return ">=";
        case ADD :              return "+";
        case SUB :              return "-";
        case MUL :              return "*";
        case DIV :              return "/";
        case AST_UNARY_PLUS :   return "+";
        case AST_UNARY_MINUS :  return "-";
        case AST_UNARY_NOT :    return "!";
        default:                return "?????????";
    }
}

char * DynOP(int ma) {
    switch(ma) {
        case AST_ASSIGN :      return "OP_ASSIGN";
        case ADD :              return "OP_ADD";
        case SUB :              return "OP_SUB";
        case MUL :              return "OP_MUL";
        case DIV :              return "OP_DIV";
        case OR :              return "OP_OR";
        case AND :              return "OP_AND";
        case EQ :              return "OP_EQ";
        case NE :              return "OP_NE";
        case LT :              return "OP_LT";
        case GT :              return "OP_GT";
        case LE :              return "OP_LE";
        case GE :              return "OP_GE";
        case AST_UNARY_PLUS :   return "OP_PLUS";
        case AST_UNARY_MINUS :  return "OP_MINUS";
        case AST_UNARY_NOT :    return "OP_NOT";
        default :              return "OP_UNKNOWN" ;
    }
}

char * typeMacro(int t) {
    switch(t) {
        case VOID_T :          return "VOID_T";
        case BOOL_T :          return "BOOL_T";
        case INT_T :           return "INT_T";
        case FLOAT_T :         return "FLOAT_T";
        case STRING_T :        return "STRING_T";
        case VLIST_T :         return "VLIST_T";
        case ELIST_T :         return "ELIST_T";
        case VERTEX_T :        return "VERTEX_T";
        case EDGE_T :          return "EDGE_T";
        case GRAPH_T :         return "GRAPH_T";
        case DYN_ATTR_T :      return "DYN_ATTR_T";
        default :              return "UNKNOWN_T";
    }
}

char * assignFunc(int t) {
    switch(t) {
        case STRING_T :        return "assign_operator_string";
        case VLIST_T :         return "assign_operator_list";
        case ELIST_T :         return "assign_operator_list";
        case VERTEX_T :        return "assign_operator_vertex";
        case EDGE_T :          return "assign_operator_edge";
        case GRAPH_T :         return "assign_operator_graph";
        case DYN_ATTR_T :      return "assign_operator_attr";
    }
}

```



```

        default :                return "XXXXXXXXXXXXX";
    }
}

char * gotolabel() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"label_%d\\0", i++);
    return tmp;
}

char * tmpReturnTmp() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"_tmp_return_%d\\0", i++);
    return tmp;
}

char * tmpMatch() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"_tmp_match_%d\\0", i++);
    return tmp;
}

char * tmpMatchStr() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"_STR_tmp_match_%d\\0", i++);
    return tmp;
}

char * tmpMatchStrVab() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"_STRV_%d\\0", i++);
    return tmp;
}

char * tmpGraphVab() {
    static char tmp[128];
    static int i = 0;
    sprintf(tmp,"_tmp_g_%d\\0", i++);
    return tmp;
}

SymbolTableEntry* tmpVab(int type, ScopeId sid) {
    static Lexeme tmp;
    static int i = 0;
    sprintf(tmp,"_tmp_vab_%d\\0", i++);
    SymbolTableEntry* e = tmpNewVarEty(tmp, type, sid);
    tmpTableInsert( e );
    return e;
}

char * tmpVabAssign( SymbolTableEntry* e, char * value ) {
    if (e->type == DYN_ATTR_T || e->type == VLIST_T ||
        e->type == ELIST_T || e->type == VERTEX_T ||
        e->type == EDGE_T || e->type == GRAPH_T || e->type == STRING_T) {
        return strCatAlloc("", 6,
            assignFunc( e->type ), " ( &( ", e->bind, " ) , ", value, " );\\n");
    }
    else {
        return strCatAlloc("", 1, "Wrong Call in tmpVabAssign()");
    }
}

char * tmpVabDel( SymbolTableEntry* e ){
    return strCatAlloc("",6, codeFreeFuncName(e->type), " ( ",
        e->bind, " );", e->bind, " = NULL;\\n");
}

void codeIndentInit(){
    int mlvl = maxLevel, i;
    //printf("Max LEVEL: %d\\n", mlvl);
    INDENT = (char **) malloc( sizeof( char * ) * (mlvl+1) );
    INDENT[0] = strCatAlloc("",1,"");
    for (i=1; i<=mlvl; ++i) {
        INDENT[i] = strCatAlloc("",2,INDENT[i-1]," ");
    }
}

```

```

void codeIndentFree() {
    int mlvl = maxLevel, i=0;
    sTableMaxLevel(&mlvl);
    for (i=0; i<=mlvl; ++i) {
        //printf("%d %d\n",i,strlen(INDENT[i]));
        free(INDENT[i]);
    }
    free(INDENT);
}

```

../src/CodeGenUtil.c

5 Logging and Reporting

```

// author : Chantal, Kunal, Lixing, Jing

#ifndef ERROR_H_NSBL_
#define ERROR_H_NSBL_

/*****
 * Internal Errors *
 *****/
#define ErrorSymbolTableKeyAlreadyExsit -301
#define ErrorNoBinderForId -302
#define ErrorNoBinderForAttribute -303

/*****
 * Compiler Error *
 *****/
// Lex 0-20
#define ErrorUnrecognizedLexeme +1

// syntax 21-50
#define ErrorSyntax +21
#define ErrorAttributeDeclaration +22
#define ErrorDerivedTypeDeclaration +23
#define ErrorAssignmentExpression +24
#define ErrorListMixedType +25
#define ErrorEdgeAssignExpression +26

// semantic 51-100
#define ErrorIdentifierAlreadyDeclared +51
#define ErrorIdentifierUsedBeforeDeclaration +52
#define ErrorFunctionCalledBeforeDeclaration +53
#define ErrorFunctionCallNOTEqualNumberOfParameters +54
#define ErrorFunctionCallIncompatibleParameterType +55
#define ErrorFuncLiteralCallIncompatibleParameterType +56
#define ErrorAttributeTypeNotSupported +57
#define ErrorDeclareAttrForWrongType +58
#define ErrorDelAttrFromWrongType +59
#define ErrorDelVariableOfWrongType +60
#define ErrorBinaryOperationWithDynamicType +61
#define ErrorGetAttrForWrongType +62
#define ErrorMactchWrongType +63
#define ErrorWrongFuncCall +64
#define ErrorInvalidReturnType +65
#define ErrorNoReturnInFunc +66
#define ErrorIfConditionNotBOOL +67
#define ErrorGetMemberForNotListType +68
#define ErrorVoidTypeVariableNotSupported +69
#define ErrorPipeWrongType +70
#define ErrorForeachType +71
#define ErrorAssignLeftOperand +72
#define ErrorAssignInMatch +73
#define ErrorWrongArgumentType +74
#define ErrorGetLengthForTypeNotList +75
#define ErrorIllegalDerivedTypeDeclaration +76
#define ErrorInitDerivedType +77
#define ErrorPrintWrongType +78
#define ErrorNestedFuncLiteralInFuncLiteral +79
#define ErrorCallBreakOutsideOfLoop +80
#define ErrorCallContinueOutsideOfLoop +81
#define ErrorCallReturnOutsideOfFunc +82
#define ErrorDynamicAttributeUsedInNonDynamicScope +83
#define ErrorOperatorNotSupportedByType +91

```

```

#define ErrorTypeMismatch                                +92
#define ErrorCastType                                  +93
/*****
 * output *
*****/
// ATTENTION: always set ERRNO before calling errorInfo
void errorInfo(int eno, long long line, char* fmt, ...);
void errorInfoExt(char* fmt, ...);
void errorInfoNote(char* fmt, ...);
#endif

```

../src/Error.h

```

// author : Jing
//
#include <stdio.h>
#include <stdarg.h>
FILE* ERRORIO;
int ERRNO;

void errorInfo(int eno, long long line, char* fmt, ...){
    va_list args;
    va_start(args, fmt);
    fprintf(ERRORIO, "ERROR:%lld:%d: ", line, eno);
    vfprintf(ERRORIO, fmt, args);
    va_end(args);
    ERRNO = eno;
    return;
}

void errorInfoExt(char* fmt, ...){
    va_list args;
    va_start(args, fmt);
    vfprintf(ERRORIO, fmt, args);
    va_end(args);
    return;
}

void errorInfoNote(char* fmt, ...){
    va_list args;
    va_start(args, fmt);
    fprintf(ERRORIO, ">>>NOTE>>>: ");
    vfprintf(ERRORIO, fmt, args);
    va_end(args);
    return;
}

```

../src/Error.c

```

// author : Jing
#ifndef UTILS_H_NSBL_
#define UTILS_H_NSBL_
#include "ASTree.h"

void init_util();
void debugInfo(char* fmt, ...);
void debugInfoExt(char* fmt, ...);
void logInfo(char* fmt, ...);

void showASTandST(struct Node* node, const char * head);
#endif

```

../src/util.h

```

// author : Jing
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "util.h"
extern SymbolTable*      s_table;
extern SymbolTable*      tmp_table;
extern SymbolTableStack* s_stack;
extern int ERRNO;

#ifdef _DEBUG
FILE* DEBUGIO;
#endif
#ifndef _NO_LOG
FILE* LOGIO;
#endif

```

```

FILE* ERRORIO;

long long LEXLINECOUNTER;

void init_util() {
#ifdef _DEBUG
    DEBUGIO = stdout;
#endif
#ifdef _NO_LOG
    LOGIO = stdout;
#endif
    ERRORIO = stderr;
    LEXLINECOUNTER = 1;
}

void debugInfo(char* fmt, ...){
#ifdef _DEBUG
    va_list args;
    va_start(args, fmt);
    fprintf(DEBUGIO, "DEBUG: ");
    vfprintf(DEBUGIO, fmt, args);
    va_end(args);
#endif
    return;
}

void debugInfoExt(char* fmt, ...){
#ifdef _DEBUG
    va_list args;
    va_start(args, fmt);
    vfprintf(DEBUGIO, fmt, args);
    va_end(args);
#endif
    return;
}

void logInfo(char* fmt, ...){
#ifdef _NO_LOG
    va_list args;
    va_start(args, fmt);
    vfprintf(LOGIO, fmt, args);
    va_end(args);
#endif
    return;
}

void showASTandST(struct Node* node, const char * head) {
#ifdef _NO_LOG
    logInfo("====AST::%s====\n", head);
    astOutTree(node, LOGIO, 0);
    logInfo("====Symbol Table====\n");
    sTableShow(LOGIO);
    logInfo("====TMP Table====\n");
    tmpTableShow(LOGIO);
#endif
}

```

../src/util.c

6 Global Variables

```

// author : Jing

#ifdef GLOBAL_H_NSBL_
#define GLOBAL_H_NSBL_
#include <stdio.h>
#include "SymbolTable.h"

/** util */
#ifdef _DEBUG
extern FILE* DEBUGIO;
#endif
#ifdef _NO_LOG
extern FILE* LOGIO;
#endif
extern long long LEXLINECOUNTER;

```

```

/** SymbolTable */
extern SymbolTable*      s_table;
extern SymbolTable*      tmp_table;
extern SymbolTableStack* s_stack;
extern int               isDynamicScope;
extern int               isNoTypeCheck;
extern int               maxLevel;

/** Error */
extern int ERRNO;
extern FILE* ERRORIO;

/** code Generation */
extern char * OUTFILE;
extern FILE * OUTFILESTREAM;
extern char ** INDENT;
extern int  inLoop, inFunc, inFuncLiteral, isFunc, inMATCH, existMATCH, nMATCHsVab, existPIPE;
extern GList *returnList, *noReturn, *FuncParaList, *returnList2, *noReturn2;
extern char * matchStaticVab, *frontDeclExp, *frontDeclExpTpl;
#endif

```

../src/global.h

7 Shared Head files

```

#ifndef NSBL_H_
#define NSBL_H_

#include "Derivedtype.h"
#include "NSBLio.h"
#include "FileReadWrite.h"
#include "GC.h"

#endif

```

../src/nsbl.h

```

// author: Jing and Lixing

#ifndef TYPE_H_NSBL
#define TYPE_H_NSBL

#define VOID_T      0
#define BOOL_T      1
#define INT_T       2
#define FLOAT_T     3
#define STRING_T    4
#define VLIST_T     5
#define ELIST_T     6
#define VERTEX_T    7
#define EDGE_T      8
#define GRAPH_T     9
#define FUNC_T      10
#define FUNC_LITERAL_T 11
#define LIST_INIT_T 12

#define DYN_BOOL_T   -1
#define DYN_INT_T    -2
#define DYN_FLOAT_T  -3
#define DYN_STRING_T -4
#define DYN_ATTR_T   -10
#define DYNAMIC_T    -11

#define UNKNOWN_T    -99
#define NOT_AVAIL    -55

#define RESERVED    -100

#endif

```

../src/type.h

```

// author : Jing

#ifndef OPERATOR_H_NSBL

```

```

#define OPERATOR_H_NSBL

#define OP_ASSIGN    0x100
#define OP_ADD       0x101
#define OP_SUB       0x102
#define OP_MUL       0x103
#define OP_DIV       0x104

#define OP_OR        0x105
#define OP_AND       0x106
#define OP_EQ        0x107
#define OP_NE        0x108
#define OP_LT        0x109
#define OP_GT        0x10A
#define OP_LE        0x10B
#define OP_GE        0x10C
#define OP_PLUS      0x10D
#define OP_MINUS     0x10E
#define OP_NOT       0x10F

#define OP_OUTE      0x110
#define OP_INE       0x111
#define OP_SV        0x112
#define OP_EV        0x113

#define OP_UNKNOWN   0x1FF

#endif

```

../src/operator.h

8 NSBL Graph library

```

// author : Lixing

#ifndef DERIVEDTYPE_H_NSBL
#define DERIVEDTYPE_H_NSBL
#include "type.h"
#include "operator.h"

#include <glib/ghash.h>
#include <glib/gstring.h>
#include <glib/glist.h>
#include <glib/garray.h>
#include <glib/gslist.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdarg.h>

#define FLAG_NO_REVERSE    0
#define FLAG_REVERSE      1
#define FLAG_KEEP_ATTR    0
#define FLAG_DESTROY_ATTR 1

typedef long int EdgeId;
typedef long int VertexId;
typedef long int GraphId;
typedef GHashTable AttributeTable;

typedef GString StringType;

typedef union {
    bool    bv;
    int     iv;
    float   fv;
    GString* sv;
} AttrValue;

typedef struct{
    long int    type;
    AttrValue   value;
} Attribute;

```

```

typedef struct{
    VertexId id;
    AttributeTable* attributes;
    //int number_of_out;
    //int number_of_in;
    GList* ings;
    GList* outEdges;
    GList* inEdges;
}VertexType;

typedef struct{
    EdgeId id;
    VertexType* start;
    VertexType* end;
    GList* ings;
    AttributeTable* attributes;
}EdgeType;

typedef struct{
    GraphId id;
    //int number_of_e;
    //int number_of_v;
    GList* edgeIdList;
    GList* vertexIdList;
    GHashTable* edges;
    GHashTable* vertices;
}GraphType;

typedef struct{
    int type;
    GList* list;
}ListType;

/*Function declaration*/
/*Init*/
EdgeType*      new_edge();
VertexType*    new_vertex();
GraphType*     new_graph();
ListType*      new_list();
StringType*    new_string();

int            destroy_edge(EdgeType* e);
int            destroy_vertex(VertexType* v);
int            destroy_graph(GraphType* g);
int            destroy_list(ListType* list);
int            destroy_string(StringType* s);

Attribute*     new_attr( int type, void * val);
Attribute*     new_attr_INT_T(int i);
Attribute*     new_attr_FLOAT_T(float f);
Attribute*     new_attr_BOOL_T(bool b);
Attribute*     new_attr_STRING_T(GString* s);

int            destroy_attr ( Attribute * attr );
int            assign_attr( Attribute * attr, int type, void * val );
int            cmp_attr( Attribute * attr1, void * val );
void           output_attr( char * key, Attribute * attr, FILE * out );
static void    destroy_attr_from_table ( gpointer key, gpointer entry, gpointer dummy2 );
void *         get_attr_value( Attribute * attr , int type, int lno);          //TODO
int            get_attr_value_INT_T(Attribute* attr, int lno);
float          get_attr_value_FLOAT_T(Attribute* attr, int lno);
bool           get_attr_value_BOOL_T(Attribute* attr, int lno);
StringType*    get_attr_value_STRING_T(Attribute* attr, int lno);

int            edge_assign_direction(EdgeType* e, VertexType* v1, VertexType* v2);
int            edge_assign_attribute(EdgeType* e, char* attribute, void* value, int type);
int            edge_remove_attribute(EdgeType* e, char* attribute);
Attribute*     edge_get_attribute(EdgeType* e, char* attribute, int autoNew, int lno);      //TODO
void*          edge_get_attribute_value(EdgeType* e, char* attribute, int lno);
VertexType*    get_end_vertex(EdgeType* e);
VertexType*    get_start_vertex(EdgeType* e);

int            vertex_assign_attribute(VertexType* v, char* attribute, void* value, int type);
int            vertex_remove_attribute(VertexType* v, char* attribute);
Attribute*     vertex_get_attribute(VertexType* v, char* attribute, int autoNew, int lno);    //TODO
void*          vertex_get_attribute_value(VertexType* v, char* attribute, int lno);
GList*         get_v_outedges(VertexType* v);
GList*         get_v_inedges(VertexType* v);

GList*         get_ving_outedges(GraphType* g, VertexType* v);
GList*         get_ving_inedges(GraphType* g, VertexType* v);

```

```

GList*      get_g_all(GraphType* g);
GList*      get_g_allv(GraphType* g);
ListType*   get_g_elist(GraphType* g);
ListType*   get_g_vlist(GraphType* g);

int          g_remove_edge(GraphType* g, EdgeType* e);
int          g_remove_vertex(GraphType* g, VertexType* v);
int          g_insert_v(GraphType* g, VertexType* v);
int          g_insert_e(GraphType* g, EdgeType* v);
int          g_insert_subg(GraphType* g, GraphType* subg);
int          g_append_list(GraphType* g, ListType* list);
void         g_free_a_vertex( gpointer key, gpointer value, gpointer dummy );
void         g_free_an_edge( gpointer key, gpointer value, gpointer dummy );
void         g_free_all_vertex( GHashTable * gh );
void         g_free_all_edge( GHashTable * gh );

ListType*    match_string(ListType* list, char* attribute, char* s);
ListType*    match_num(ListType* list, char* attribute, float a, int op);
ListType*    pipe(ListType* list, int pipiop);

ListType*    list_declaration(int type, int n, ...);
void*        list_getelement(ListType* list, int index);
ListType*    list_append(ListType* list, int type, void* obj);
int          list_assign_element(ListType* list, int type, int index, void* obj);
ListType*    list_append_gl(ListType *l, GList* gl, int type);

/*print functions*/
int          print_g(GraphType* g);
int          print_v(VertexType* v);
int          print_e(EdgeType* e);
int          print_v_attr(VertexType* v);
int          print_e_attr(EdgeType* e);
int          print_LIST_T(ListType* l);
#define print_ELIST_T print_LIST_T
#define print_VLIST_T print_LIST_T

int          print_VERTEX_T(VertexType* v);
int          print_EDGE_T(EdgeType* e);
int          print_GRAPH_T(GraphType* g);
int          print_attr(Attribute* attr);

//TODO
Attribute*   binary_operator( Attribute* attr1, Attribute* attr2, int op, int reverse, int rm_attr1, int
    rm_attr2, int lno);
// static = attr1
void         assign_operator_to_static( Attribute* attr1, int type, void * value, int rm_attr1, int lno);
// attr1 = attr2
Attribute*   assign_operator( Attribute* attr1, Attribute* attr2, int rm_attr1, int rm_attr2, int lno);
Attribute*   unary_operator( Attribute* attr1, int op, int rm_attr1, int lno);
Attribute*   cast_operator( Attribute* attr1, int type, int rm_attr1, int lno);
ListType*    list_match( ListType * l, bool (*func) (void *, int ), int rm_l );
ListType*    list_pipe(ListType* l, int type, int pipe_op, int rm_l);
Attribute*   object_get_attribute(void* v, int obj, char* attribute, int autoNew, int lno);

// DONE
StringType*  assign_operator_string(StringType** s1, StringType* s2);
ListType*    assign_operator_list(ListType** l1, ListType* l2);
VertexType*  assign_operator_vertex(VertexType** v1, VertexType* v2);
EdgeType*    assign_operator_edge(EdgeType** e1, EdgeType* e2);
GraphType*   assign_operator_graph(GraphType** gl, GraphType* g2);
Attribute*   assign_operator_attr(Attribute** a1, Attribute* a2);
void         die(int lno, char* fmt, ...);

//int list_append(ListType* list, void* data);
//ListType* list_declare(...);
//int list_remove(void* data);

//StringType* string_append(StringType* s, const char* seq);
#endif

```

../src/Derivedtype.h

```

/*****
- All Except blow :      Lixing
- GC related/debug info : Jing
*****/
#include "Derivedtype.h"
#include <string.h>

```



```

#include <stdlib.h>
#include <string.h>
#include "GC.h"

#define LIST_T VLIST_T
// may have problem when delete a ...
EdgeId new_edgeId(){
    static EdgeId eid = 0;
    return eid++;
}

VertexId new_vertexId(){
    static VertexId vid = 0;
    return vid++;
}

GraphId new_graphId(){
    static GraphId gid = 0;
    return gid++;
}

//key1 needs to be a pointer to an int
int g_hash_table_contains(GHashTable* t, void* key1){
    if(t==NULL || key1==NULL)
        return 0;
    GList* list = g_hash_table_get_keys(t);
    int l = g_list_length(list);
    int n = 0;
    for(n; n<l; n++){
        void* key2 = g_list_nth_data(list,n);
        if(*(int*)key1 == *(int*)key2){
            g_list_free(list);
            return 1;
        }
    }
    g_list_free(list);
    return 0;
}

EdgeType* new_edge(){
    EdgeType* edge = (EdgeType*) malloc(sizeof(EdgeType));
    if(edge == NULL)
        die(-1, "memory allocation failed in function: new_edge()\n");
    edge->id = new_edgeId();
    edge->start = NULL;
    edge->end = NULL;
    edge->ings = NULL;
    edge->attributes = g_hash_table_new(g_str_hash, g_str_equal);
    return edge;
}

VertexType* new_vertex(){
    VertexType* vertex = (VertexType*) malloc(sizeof(VertexType));
    if(vertex == NULL)
        die(-1, "memory allocation failed in function: new_vertex()\n");
    vertex->id = new_vertexId();
    vertex->attributes = g_hash_table_new(g_str_hash, g_str_equal);
    vertex->outEdges = NULL;
    vertex->inEdges = NULL;
    vertex->ings = NULL;
    return vertex;
}

GraphType* new_graph(){
    GraphType* graph = (GraphType*) malloc(sizeof(GraphType));
    if(graph==NULL)
        die(-1, "memory allocation failed in function: new_graph()\n");
    graph->id = new_graphId();
    graph->edgeIdList = NULL;
    graph->vertexIdList = NULL;
    graph->edges = g_hash_table_new(g_str_hash, g_str_equal);
    graph->vertices = g_hash_table_new(g_int_hash, g_int_equal);
    return graph;
}

ListType* new_list(){
    ListType* l = (ListType*)malloc(sizeof(ListType));
    if(l==NULL)
        die(-1, "memory allocation failed in function: new_list()\n");
    l->type = UNKNOWN_T;
    l->list = NULL;
}

```

```

    return 1;
}

StringType* new_string(){
    return (StringType*)g_string_new("");
}

int destroy_edge(EdgeType* e){
    if(e==NULL) return 0;
    // GC
    int nref;
    if ( (nref=gcDef( (void *) e, EDGE_T )) > 0 ) return nref;
#ifdef _DEBUG
    fprintf(stdout, "DEBUG: DESTROY EDGE: ");
    print_e(e);
    fprintf(stdout, "\n");
    fprintf(stdout, "----- REMOVE ATTR\n");
#endif
    if(g_hash_table_size(e->attributes) > 0)
        g_hash_table_foreach(e->attributes, &destroy_attr_from_table, NULL);
    g_hash_table_destroy(e->attributes);
#ifdef _DEBUG
    fprintf(stdout, "----- REMOVE me FROM v1 and v2\n");
#endif
    /*
    VertexType* v1 = e->start;
    VertexType* v2 = e->end;
    if(v1 != NULL) v1->outEdges = g_list_remove(v1->outEdges, e);
    if(v2 != NULL) v2->inEdges = g_list_remove(v2->inEdges, e);
    e->start = NULL;
    e->end = NULL;
    */ //bug !!!
#ifdef _DEBUG
    fprintf(stdout, "----- REMOVE me FROM all Gs\n");
#endif
    /*
    int l = g_list_length(e->ings);
    int n = 0;
    for(n; n<l; n++){
        GraphType* g = g_list_nth_data(e->ings, n);
        g_remove_edge(g, e);
    }
    */
#ifdef _DEBUG
    fprintf(stdout, "----- REMOVE INGS\n");
#endif
    g_list_free(e->ings);
    free(e);
    return 0;
}

int destroy_vertex(VertexType* v){
    if(v==NULL) return 0;
    // GC
    int nref;
    if ( (nref = gcDef( (void *) v, VERTEX_T )) > 0 ) return nref;
#ifdef _DEBUG
    fprintf(stdout, "DEBUG: DESTROY VERTEX: ");
    print_v(v);
    fprintf(stdout, "\n");
    fprintf(stdout, "===== REMOVE ALL ATTR\n");
#endif
    g_hash_table_foreach(v->attributes, &destroy_attr_from_table, NULL);
    g_hash_table_destroy(v->attributes);
    EdgeType* e;
    int l = g_list_length(v->outEdges);
    int n = 0;
    while ( g_list_length(v->outEdges) > 0 ){
#ifdef _DEBUG
        fprintf(stdout, "===== REMOVE OUTEdges %d/%d\n", n++, l );
#endif
        e = (EdgeType*) g_list_nth_data(v->outEdges, 0);
        v->outEdges = g_list_remove( v->outEdges, e );
    }
    l = g_list_length(v->inEdges);
    n = 0;
    while ( g_list_length(v->inEdges) > 0 ) {
#ifdef _DEBUG
        fprintf(stdout, "===== REMOVE INEdges %d/%d\n", n++, l );
#endif
        e = (EdgeType*) g_list_nth_data(v->inEdges, 0);
        v->inEdges = g_list_remove( v->inEdges, e );
    }
}

```

```

    l = g_list_length(v->ings);
#ifdef _DEBUG
    fprintf(stdout, "==== REMOVE me FROM ALL Gs %d/%d\n", n, l );
#endif
/*    for(n=0; n<l; n++){
        GraphType* g = g_list_nth_data(v->ings, n);
        g_remove_vertex(g, v);
    }*/
#ifdef _DEBUG
    fprintf(stdout, "==== REMOVE MYSELF: outE, inE, ings\n");
#endif
g_list_free(v->outEdges);
g_list_free(v->inEdges);
g_list_free(v->ings);
free(v);
return 0;
}

int destroy_graph(GraphType* g){
    if(g==NULL) return 0;
    // GC
    int nref;
    if ( (nref = gcDef( (void *) g, GRAPH_T )) > 0) return nref;
    g_list_free_1(g->edgeIdList);
    g_list_free_1(g->vertexIdList);
    //g_hash_table_foreach(g->vertices, &g_free_a_vertex, NULL);
    //g_hash_table_foreach(g->edges, &g_free_a_vertex, NULL);
    g_free_all_edge( g->edges );
    g_free_all_vertex( g->vertices );
    g_hash_table_destroy(g->edges);
    g_hash_table_destroy(g->vertices);
    free(g);
    return 0;
}

void g_free_all_vertex( GHashTable * gh ) {
    GList * gl = g_hash_table_get_values ( gh );
    int l = g_list_length( gl );
    int i;
    for (i=0; i<l; i++) {
        VertexType * v = (VertexType *) g_list_nth_data( gl, i );
        destroy_vertex( v );
    }
    g_list_free( gl );
}

void g_free_all_edge( GHashTable * gh ) {
    GList * gl = g_hash_table_get_values ( gh );
    int l = g_list_length( gl );
    int i;
    for (i=0; i<l; i++) {
        EdgeType * e = (EdgeType *) g_list_nth_data( gl, i );
        destroy_edge( e );
    }
    g_list_free( gl );
}

// glib bug : g_hash_table_foreach
void g_free_a_vertex( gpointer key, gpointer value, gpointer dummy ) {
    printf("%d\n", *(int*) key);
    VertexType * v = (VertexType *) value;
    destroy_vertex( v );
}

void g_free_an_edge( gpointer key, gpointer value, gpointer dummy ) {
    destroy_edge( (EdgeType *) value );
}

int destroy_list(ListType* list){
    if(list == NULL) return 0;
    // GC
    int nref;
    if ( (nref = gcDef( (void *) list, LIST_T )) > 0) return nref;
    g_list_free(list->list);
    free(list);
    return 0;
}

int destroy_string(StringType* s){
    if(s == NULL)return 0;
    // GC

```

```

    int nref;
    if ( (nref = gcDef( (void *) s, STRING_T )) > 0) return nref;
    g_string_free((GString*)s, 1);
    return 0;
}

int edge_assign_direction(EdgeType* e, VertexType* v1, VertexType* v2){
    if(e == NULL)
        die(-1, "assign direction to a NULL edge in function: edge_assign_direction\n");
    if(v1==NULL || v2==NULL)
        die(-1, "assign direction for NULL vertex in function: edge_assign_direction\n");
    e->start = v1;
    e->end = v2;
    v1->outEdges = g_list_append(v1->outEdges, e);
    v2->inEdges = g_list_append(v2->inEdges, e);
    return 0;
}

// IMPORTANT : all the other initializor of attr must be wapper of this one,
// for GC
Attribute * new_attr( int type, void * val ) {
    Attribute * attr = (Attribute *) malloc ( sizeof( Attribute ) );
    attr->type = type;
    switch (type) {
        case INT_T :
            attr->value.iv = (val==NULL) ? 0 : *(int *)val; break;
        case FLOAT_T :
            attr->value.fv = (val==NULL) ? 0.0 : *(float *)val; break;
        case BOOL_T:
            attr->value.bv = (val==NULL) ? 0 : ((* (bool*)val > 0) ? true : false); break;
        case STRING_T :
            attr->value.sv = (val==NULL) ? NULL : val; break;
        case UNKNOWN_T :
            attr->value.sv = NULL;
            break;
        default :
            fprintf(stderr, "Derivedtype:new_attr: unknown type!!!\n");
    }
    return attr;
}

Attribute* new_attr_INT_T(int i){
    return new_attr(INT_T, &i);
}

Attribute* new_attr_FLOAT_T(float f){
    return new_attr(FLOAT_T, &f);
}

Attribute* new_attr_BOOL_T(bool b){
    return new_attr(BOOL_T, &b);
}

Attribute* new_attr_STRING_T(GString* s){
    return new_attr(STRING_T, s);
}

int assign_attr( Attribute * attr, int type, void * val ) {
    if(attr == NULL)
        die(-1, "assign value to a NULL attribute in function: assign_attr()\n");
    if(val == NULL)
        die(-1, "assign NULL value to an attribute in function: assign_attr()\n");
    switch (type) {
        case INT_T :
            attr->value.iv = * (int *) val; break;
        case FLOAT_T :
            attr->value.fv = * (float *) val; break;
        case BOOL_T:
            attr->value.bv = * (bool *) val; break;
        case STRING_T :
            if (attr->value.sv != NULL) g_string_free(attr->value.sv,1);
            attr->value.sv = (GString *) val;
            break;
        default :
            fprintf(stderr, "Derivedtype:assign_attr: unknown type!!!\n");
    }
}

int cmp_attr( Attribute * attr1, void * val ) {
    if(attr1==NULL || val==NULL)
        die(-1, "compare to NULL in function: cmp_attr()\n");
}

```

```

switch (attr1->type) {
case INT_T :
    return attr1->value.iv - * (int *) val;
case FLOAT_T : {
    float tt = attr1->value.fv - * (float *) val;
    if ( tt == 0.0 ) return 0;
    else if ( tt < 0.0 ) return -1;
    else return 1;
}
case BOOL_T:
    return (attr1->value.bv == *(bool*)val) ? 0 : 1;
case STRING_T :
    return strcmp( attr1->value.sv->str, ((GString*) val)->str );
default :
    fprintf(stderr, "Derivedtype:cmp_attr: unknown type!!!\n");
}
}

void output_attr( char * key, Attribute * attr, FILE * out ){
    if(attr==NULL)
        die(-1, "NULL attr in function: output_attr()\n");
    if(key==NULL)
        die(-1, "NULL key in function: output_attr()\n");
    if(out==NULL)
        die(-1, "NULL file pointer out in function: output_attr()\n");
    switch (attr->type) {
case INT_T :
        fprintf(out, "%s -> %d", key, attr->value.iv); break;
case FLOAT_T :
        fprintf(out, "%s -> %f", key, attr->value.fv); break;
case BOOL_T:
        fprintf(out, (attr->value.bv ? "%s -> TRUE" : "%s -> FALSE"), key); break;
case STRING_T :
        if(attr->value.sv == NULL) {
            fprintf(stderr, "output_attr: NULL String.\n");
        }
        fprintf(out, "%s -> \"%s\"", key, (attr->value.sv->str)); break;
default :
        fprintf(stderr, "Derivedtype:output_attr: unknown type %ld!!!\n", attr->type);
    }
    printf("\n");
}

int destroy_attr ( Attribute * attr ) {
    if(attr == NULL) return 0;
    // GC
    int nref;
    if ( (nref = gcDef( (void *) attr, DYN_ATTR_T )) > 0 ) return nref;
#ifdef _DEBUG
    fprintf(stderr, "DEBUG: Destroy Attr : ");
    switch ( attr->type ) {
case INT_T :
        fprintf(stderr, " INT_T --> %d\n", attr->value.iv); break;
case FLOAT_T :
        fprintf(stderr, " FLOAT_T --> %f\n", attr->value.fv); break;
case BOOL_T:
        fprintf(stderr, (attr->value.bv ? "BOOL_T --> TRUE\n" : "FLOAT --> FALSE\n")); break;
case STRING_T :
        fprintf(stderr, " STRING_T --> %s\n", attr->value.sv->str); break;
    }
#endif
    if ( attr->type == STRING_T ) g_string_free( attr->value.sv, 1 );
    free( attr );
}

static void destroy_attr_from_table ( gpointer key, gpointer entry, gpointer dummy2 ) {
    if(entry == NULL) return;
    Attribute * attr = (Attribute *) entry;
#ifdef _DEBUG
    char * attr_name = (char *) key;
    fprintf(stderr, "DEBUG: Remove attr '%s' from table \n", attr_name);
#endif
    destroy_attr( attr );
}

int get_attr_value_INT_T(Attribute* attr, int lno) {
    if(attr == NULL) die(lno, "get_attr_value_INT_T: null attribute.\n");
    if(attr->type == INT_T)
        return attr->value.iv;
    else if (attr->type == FLOAT_T)

```

```

        return (int) attr->value.fv;
    else
        die(lno, "get_attr_value_INT_T: attribute type NOT INT or FLOAT\n");
}

float get_attr_value_FLOAT_T(Attribute* attr, int lno) {
    if(attr == NULL) die(lno, "get_attr_value_FLOAT_T: null attribute.\n");
    if(attr->type == INT_T)
        return (float) attr->value.iv;
    else if (attr->type == FLOAT_T)
        return attr->value.fv;
    else
        die(lno, "get_attr_value_FLOAT_T: attribute type NOT INT or FLOAT\n");
}

bool get_attr_value_BOOL_T(Attribute* attr, int lno) {
    if(attr == NULL) die(lno, "get_attr_value_BOOL_T: null attribute.\n");
    if(attr->type == BOOL_T)
        return attr->value.bv;
    else{
        printf("attr_type: %ld\n", attr->type);
        die(lno, "get_attr_value_BOOL_T: attribute type NOT BOOL.\n");
    }
}

StringType* get_attr_value_STRING_T(Attribute* attr, int lno) {
    if(attr == NULL) die(lno, "get_attr_value_STRING_T: null attribute.\n");
    if(attr->type == STRING_T)
        return g_string_new(attr->value.sv->str);
    else
        die(lno, "get_attr_value_STRING_T: attribute type NOT STRING.\n");
}

void* get_attr_value(Attribute* attr, int type, int lno){
    if(attr == NULL) die(lno, "get_attr_value: <null> Attribute error \n");
    void * rt = NULL;
    switch(attr->type){
        case INT_T:
            rt = (void *) &(attr->value.iv);
            break;
        case FLOAT_T:
            rt = (void *) &(attr->value.fv);
            break;
        case STRING_T:
            rt = (void *) attr->value.sv;
            break;
        case BOOL_T:
            rt = (void *) &(attr->value.bv);
            break;
        default:
            return NULL;
    }
    if(type != attr->type && type != RESERVED)
        die(lno, "get_attr_value: attribute type mismatch : %d != %d \n", type, attr->type);
    return rt;
}

int edge_assign_attribute ( EdgeType* e, char * attr_name, void * val, int type ) {
    if(e==NULL)
        die(-1, "NULL edge in function: edge_assign_attribute()\n");
    if(attr_name==NULL)
        die(-1, "NULL attr_name in function: edge_assign_attribute()\n");
    Attribute* attr = (Attribute*)g_hash_table_lookup(e->attributes, attr_name);
    if (attr != NULL) {
        if ( attr->type != type ) {
            die(-1, "edge_assign_attrbute: attribute type mismatch error \n");
        }
    }
    else {
        attr = new_attr( type, NULL );
        g_hash_table_insert( e->attributes, attr_name, attr );
        // GC : see vertex_assign_attribute
        gcRef( (void *) attr, DYN_ATTR_T );
    }
    assign_attr( attr, type, val );
    return 0;
}

int edge_remove_attribute(EdgeType* e, char* attr_name){
    if (e == NULL) {
#ifdef _DEBUG
        fprintf(stderr, "Warning: edge_remove_attribute: Edge is NULL.\n");
#endif
    }
}

```

```

#endif
    return 1;
}
if (!g_hash_table_remove(e->attributes, attr_name)) {
    die(-1, "edge_remove_attribute FAILURE to remove %s.\n", attr_name);
    return 2;
}
return 0;
}

Attribute* edge_get_attribute(EdgeType* e, char* attribute, int autoNew, int lno){
    if(e==NULL)
        die(lno, "NULL edge in function: edge_get_attribute()\n");
    Attribute* attr = g_hash_table_lookup(e->attributes, attribute);
    if (attr == NULL && autoNew) {
        attr = new_attr(UNKNOWN_T, NULL);
        g_hash_table_insert( e->attributes, attribute, attr );
        // GC
        gcRef( (void *) attr, DYN_ATTR_T );
    }
    return attr;
}

void* edge_get_attribute_value(EdgeType* e, char* attribute, int lno){
    if(e==NULL)
        die(lno, "NULL edge in function: edge_get_attribute_value()\n");
    Attribute* attr;
    if( (attr = edge_get_attribute(e, attribute, 0, lno)) != NULL)
        return get_attr_value(attr, RESERVED,lno);
    return NULL;
}

VertexType* get_start_vertex(EdgeType* e){
    if(e==NULL)
        die(-1, "NULL edge in function: get_start_vertex()\n");
    return e->start;
}

VertexType* get_end_vertex(EdgeType* e){
    if(e==NULL)
        die(-1, "NULL edge in function: get_end_vertex()\n");
    return e->end;
}

int vertex_assign_attribute(VertexType* v, char* attr_name, void * val, int type ) {
    if(v==NULL)
        die(-1, "NULL vertex in function: vertex_assign_attribute()\n");
    if(attr_name==NULL)
        die(-1, "NULL attr_name in function: vertex_assign_attribute()\n");
    Attribute* attr = (Attribute*)g_hash_table_lookup(v->attributes, attr_name);
    if (attr != NULL) {
        if ( attr->type != type ) {
            die(-1, "vertex_assign_attribute: attribute type mismatch error \n");
        }
    }
    else {
        attr = new_attr( type, NULL );
        g_hash_table_insert( v->attributes, attr_name, attr );
        // GC: in FileReadWrite.c, when we read a graph from xml, need to append info
        // to GC
        gcRef( (void *) attr, DYN_ATTR_T );
    }
    assign_attr( attr, type, val );
    return 0;
}

int vertex_remove_attribute(VertexType* v, char* attr_name) {
    if (v == NULL) {
#ifdef _DEBUG
        fprintf(stderr, "Warning: vertex_remove_attribute: Vertex is NULL.\n");
#endif
        return 1;
    }
    if (!g_hash_table_remove(v->attributes, attr_name)) {
        die(-1, "vertex_remove_attribute FAILURE to remove %s.\n", attr_name);
        return 2;
    }
    return 0;
}

Attribute* vertex_get_attribute(VertexType* v, char* attribute, int autoNew, int lno){

```

```

if(v==NULL)
    die(lno, "NULL vertex in function: vertex_get_attribute()\n");
Attribute* attr = g_hash_table_lookup(v->attributes, attribute);
if (attr == NULL && autoNew) {
    attr = new_attr(UNKNOWN_T, NULL);
    g_hash_table_insert( v->attributes, attribute, attr );
    // GC
    gcRef( (void *) attr, DYN_ATTR_T );
}
return attr;
}

void* vertex_get_attribute_value(VertexType* v, char* attribute, int lno){
    if(v==NULL)
        die(lno, "NULL vertex in function: vertex_get_attribute_value()\n");
    Attribute* attr;
    if( (attr = vertex_get_attribute(v, attribute, 0, lno)) != NULL)
        return get_attr_value(attr, RESERVED,lno);
    return NULL;
}

GList* get_v_outedges(VertexType* v){
    if(v==NULL)
        die(-1, "NULL vertex in function: get_v_outedges()\n");
    return v->outEdges;
}

GList* get_v_inedges(VertexType* v){
    if(v==NULL)
        die(-1, "NULL vertex in function: get_v_inedges()\n");
    return v->inEdges;
}

GList* get_common_edges(GHashTable* edges, GList* list){
    GList* common = NULL;
    if(edges==NULL || list==NULL)
        return common;
    int l = g_list_length(list);
    int n = 0;
    for(n; n<l; n++){
        EdgeType* e = g_list_nth_data(list, n);
        if(g_hash_table_contains(edges, &(e->id)))
            common = g_list_append(common, e);
    }
    return common;
}

GList* get_ving_outedges(GraphType* g, VertexType* v){
    if(g==NULL || v==NULL)
        return NULL;
    GList* elist = get_v_outedges(v);
    return get_common_edges(g->edges, elist);
}

GList* get_ving_inedges(GraphType* g, VertexType* v){
    if(g==NULL || v==NULL)
        return NULL;
    GList* elist = get_v_inedges(v);
    return get_common_edges(g->edges, elist);
}

GList* get_g_allv(GraphType* g){
    if(g==NULL)
        return NULL;
    GList* list = NULL;
    int l = g_list_length(g->vertexIdList);
    int n = 0;
    for(n; n<l; n++){
        int* key = g_list_nth_data(g->vertexIdList, n);
        VertexType* v = (VertexType*)g_hash_table_lookup(g->vertices, key);
        list = g_list_append(list, v);
    }
    return list;
}

ListType* get_g_vlist(GraphType* g){
    ListType* lt = new_list();
    lt->type = VERTEX_T;
    lt->list = get_g_allv(g);
    return lt;
}

```



```

GList* get_g_all(GraphType* g){
    if(g==NULL)
        return NULL;
    GList* list = NULL;
    int l = g_list_length(g->edgeIdList);
    int n = 0;
    for(n; n<l; n++){
        int* key = g_list_nth_data(g->edgeIdList, n);
        EdgeType* e = (EdgeType*)g_hash_table_lookup(g->edges, key);
        list = g_list_append(list, e);
    }
    return list;
}

ListType* get_g_elist(GraphType* g){
    ListType* lt = new_list();
    lt->type = EDGE_T;
    lt->list = get_g_all(g);
    return lt;
}

int g_remove_edge(GraphType* g, EdgeType* e){
    if(g==NULL || e==NULL)
        return 0;
    g->edgeIdList = g_list_remove(g->edgeIdList, &(e->id));
    g_hash_table_remove(g->edges, e);
    return 0;
}

int g_remove_vertex(GraphType* g, VertexType* v){
    if(g==NULL || v==NULL)
        return 0;
    g->vertexIdList = g_list_remove(g->vertexIdList, &(v->id));
    g_hash_table_remove(g->vertices, v);
    return 0;
}

int g_insert_v(GraphType* g, VertexType* v){
    if(g==NULL || v==NULL) return 0;
    if (g_hash_table_lookup(g->vertices, &(v->id))==NULL) {
        g->vertexIdList = g_list_append(g->vertexIdList, &(v->id));
        g_hash_table_insert(g->vertices, &(v->id), (void*) v);
        v->ings = g_list_append(v->ings, g);
        // GC
        gcRef( (void *) v, VERTEX_T );
    }
    return 0;
}

int g_insert_e(GraphType* g, EdgeType* e){
    if(g==NULL || e==NULL) return 0;
    if (g_hash_table_lookup(g->edges, &(e->id))==NULL) {
        g->edgeIdList = g_list_append(g->edgeIdList, &(e->id));
        g_hash_table_insert(g->edges, &(e->id), e);
        e->ings = g_list_append(e->ings, g);
        // GC
        gcRef( (void *) e, EDGE_T );
    }
    return 0;
}

int g_append_list(GraphType* g, ListType* list){
    if(g==NULL || list==NULL)
        return 0;
    int length = g_list_length(list->list);
    int i;
    for(i=0; i<length; i++){
        switch(list->type){
            case VERTEX_T:
                g_insert_v(g, (VertexType*)g_list_nth_data(list->list, i));
                break;
            case EDGE_T:
                g_insert_e(g, (EdgeType*)g_list_nth_data(list->list, i));
                break;
            default:
                break;
        }
    }
    return 0;
}

```

```

}

int g_insert_subg(GraphType* g, GraphType* subg){
    if(g==NULL || subg==NULL)
        return 0;
    int l,n;
    l = g_list_length(subg->vertexIdList);
    for(n=0; n<l; n++){
        int* key = g_list_nth_data(subg->vertexIdList, n);
        if(g_hash_table_contains(subg->vertices, key))
            continue;
        else{
            g->vertexIdList = g_list_append(g->vertexIdList, key);
            VertexType* v = g_hash_table_lookup(subg->vertices, key);
            g_hash_table_insert(g->vertices, key, v);
        }
    }
    l = g_list_length(subg->edgeIdList);
    for(n=0; n<l; n++){
        int* key = g_list_nth_data(subg->edgeIdList, n);
        if(g_hash_table_contains(subg->edges, key))
            continue;
        else{
            g->edgeIdList = g_list_append(g->edgeIdList, key);
            EdgeType* e = g_hash_table_lookup(subg->edges, key);
            g_hash_table_insert(g->edges, key, e);
        }
    }
}

ListType* match_string(ListType* list, char* attribute, char* s){
    GList* result=NULL;
    if(list==NULL)
        die(-1, "NULL list in function match_string()\n");
    if(attribute==NULL || s==NULL){
        list->list = NULL;
        return list;
    }
    int l = g_list_length(list->list);
    int n = 0;
    switch(list->type){
        case EDGE_T:
            for(n; n<l; n++){
                EdgeType* e = g_list_nth_data(list->list, n);
                Attribute* attr_v;
                if((attr_v = g_hash_table_lookup(e->attributes, attribute))!=NULL){
                    if(strcmp(attr_v->value.sv->str, (char*)s)==0)
                        result = g_list_append(result, e);
                }
            }
            break;
        case VERTEX_T:
            for(n; n<l; n++){
                VertexType* e = g_list_nth_data(list->list, n);
                Attribute* attr_v;
                if((attr_v = g_hash_table_lookup(e->attributes, attribute))!=NULL){
                    if(strcmp(attr_v->value.sv->str, (char*)s)==0)
                        result = g_list_append(result, e);
                }
            }
            break;
    }
    g_list_free(list->list);
    list->list = result;
    return list;
}

ListType* match_num(ListType* list, char* attribute, float cmpv, int op){
    GList* result=NULL;
    if(list==NULL)
        die(-1, "NULL list in function match_num()\n");
    if(attribute==NULL){
        list->list = NULL;
        return list;
    }
    int l = g_list_length(list->list);
    int n = 0;
    void* e;
    for(n; n<l; n++){
        Attribute* attr_v;
        if(list->type == EDGE_T){

```

```

    EdgeType* p = (EdgeType*)g_list_nth_data(list->list, n);
    e = p;
    attr_v = g_hash_table_lookup(p->attributes, attribute);
}
else if(list->type == VERTEX_T){
    VertexType* p = (VertexType*)g_list_nth_data(list->list, n);
    e = p;
    attr_v = g_hash_table_lookup(p->attributes, attribute);
}
else
    break;
if(attr_v!=NULL){
    float f=0.0;
    if(attr_v->type == INT_T)
        f = (float)attr_v->value.iv;
    else if(attr_v->type == FLOAT_T)
        f =attr_v->value.fv;
    switch(op){
        case OP_EQ:
            if(f == cmpv)
                result = g_list_append(result, e);
            break;
        case OP_GT:
            if(f > cmpv)
                result = g_list_append(result, e);
            break;
        case OP_LT:
            if(f < cmpv)
                result = g_list_append(result, e);
            break;
        case OP_GE:
            if(f >= cmpv)
                result = g_list_append(result, e);
            break;
        case OP_LE:
            if(f <= cmpv)
                result = g_list_append(result, e);
            break;
    }
}
}
g_list_free(list->list);
list->list = result;
return list;
}

ListType* list_declaration(int type,int n, ...){
    ListType* newlist = new_list();
    if(newlist==NULL) die(-1, "failed to allocate memory for newlist in function: list_declaration()\n");
    newlist->type = type;
    int i;
    va_list args;
    va_start(args, n);
    for(i=0; i<n; i++){
        switch(type){
            case VERTEX_T:
                {VertexType* v = va_arg(args, VertexType*);
                newlist->list = g_list_append(newlist->list, v);
                }
            break;
            case EDGE_T:
                {EdgeType* e = va_arg(args, EdgeType*);
                newlist->list = g_list_append(newlist->list, e);
                }
            break;
            default:
                break;
        }
    }
    va_end(args);
    return newlist;
}

void* list_getelement(ListType* list, int index){
    if(list==NULL)
        die(-1, "list is NULL in function: list_getelement()\n");
    void * rlt = (void *) g_list_nth_data(list->list, index);
    if (rlt == NULL) die(-1,"list_getelement: member NOT exist.\n");
    return g_list_nth_data(list->list, index);
}

```

```

ListType* list_append(ListType* list, int type, void* obj){
    if(list == NULL)
        die(-1, "list is NULL in function: list_append()\n");
    if(obj==NULL)
        return list;
    if(list->type == UNKNOWN_T)
        list->type = type;
    else if(list->type != type){
        die(-1, "unmatched list append element\n");
    }
    if(g_list_index(list->list, obj) == -1){
        list->list = g_list_append(list->list, obj);
    }
    return list;
}

int list_assign_element(ListType* list, int type, int index, void* obj){
    if(list == NULL)
        die(-1, "list is NULL in function: list_assign_element()\n");
    if(obj == NULL)
        return 1; // list?
    if(g_list_length(list->list)<=(index+1))
        return 1;
    if(list->type == UNKNOWN_T)
        list->type = type;
    else if(list->type != type){
        return 1;
    }
    void* p = g_list_nth_data(list->list, index);
    p = obj;
    return 0;
}

ListType* list_append_gl(ListType* l, GList* gl, int type){
    if(l==NULL)
        die(-1, "list l is NULL if function: list_append_gl()\n");
    if(l->type != type){
        die(-1, "unmatched type for list append glist\n");
    }
    int len = g_list_length(gl);
    int i;
    void* obj;
    for(i=0; i<len; i++){
        obj = g_list_nth_data(gl,i);
        list_append(l, type, obj);
    }
    return l;
}

int print_list(ListType* list){
    if(list==NULL)
        die(-1, "list is NULL if function: print_list()\n");
    int i;
    int type = list->type;
    int length = g_list_length(list->list);
    printf("<List>: ");
    for(i=0; i<length; i++){
        switch(type){
            case VERTEX_T:
                print_v((VertexType*)g_list_nth_data(list->list, i));
                break;
            case EDGE_T:
                print_e((EdgeType*)g_list_nth_data(list->list, i));
                break;
            default: die(-1, "print_list: list print wrong type\n");
                break;
        }
        printf(" ");
    }
    printf("\n");
    return 0;
}

int print_v(VertexType* v){
    if(v==NULL)
        return 0;
    printf("<Vertex: %ld>", v->id);
    return 0;
}

int print_e(EdgeType* e){

```

```

if (e==NULL)
    return 0;
printf("<Edge: %ld>", e->id);
if (e->start!=NULL)
    printf("vstart%ld-->", e->start->id);
else
    printf("vstart[NULL]-->");
if (e->end!=NULL)
    printf("vend[%ld]})", e->end->id);
else
    printf("vend[NULL]"));
    return 0;
}

int print_v_attr(VertexType* v){
    if( v == NULL )
        die(-1, "print_v_attr: NULL pointer.\n");
    GList* klist = g_hash_table_get_keys(v->attributes);
    int l = g_list_length(klist);
    int n = 0;
    printf("\nVertex Attributes:-----\n");
    printf("<Vertex> : Id = %ld\n", v->id);
    for(n; n<l; n++){
        void* key = g_list_nth_data(klist, n);
        Attribute* value = g_hash_table_lookup(v->attributes, key);
        output_attr( (char *) key, value, stdout);
    }
    printf("-----\n");
    g_list_free(klist);
    return 0;
}

int print_e_attr(EdgeType* e){
    if( e == NULL )
        die(-1, "print_e_attr: NULL pointer.\n");
    GList* klist = g_hash_table_get_keys(e->attributes);
    int l = g_list_length(klist);
    int n = 0;
    printf("\nEdge Attributes:-----\n");
    printf("<Edge> : Id = %ld \n", e->id);
    printf("vstart");
    if (e->start!=NULL){
        printf("[");
        print_v(e->start);
        printf("]");
    }
    else
        printf("[NULL]");

    printf("-->vend");

    if (e->end!=NULL){
        printf("[");
        print_v(e->end);
        printf("]");
    }
    else
        printf("[NULL]");
    printf("\n");
    printf("-----\n");
    for(n; n<l; n++){
        void* key = g_list_nth_data(klist, n);
        Attribute* value = g_hash_table_lookup(e->attributes, key);
        output_attr( (char *) key, value, stdout);
    }
    g_list_free(klist);
    return 0;
}

int print_g(GraphType* g){
    if (g==NULL)
        return 0;
    GList* vlist = get_g_allv(g);
    GList* elist = get_g_alleg(g);
    int l,n;
    printf("\nGraph-----\n");
    l = g_list_length(vlist);
    printf("Vertices: \n");
    for(n=0; n<l; n++){
        VertexType* v = g_list_nth_data(vlist, n);
        print_v(v);
    }
}

```

```

        printf(" | ");
    }
    printf("\n");

    l = g_list_length(elist);
    printf("Edges: \n");
    for(n=0; n<l; n++){
        EdgeType* e = g_list_nth_data(elist, n);
        print_e(e);
        printf(" | ");
    }
    printf("\n");
    printf("-----\n");
    g_list_free(vlist);
    g_list_free(elist);
    return 0;
}

int print_LIST_T(ListType* l){
    if(l==NULL)
        return 0;
    print_list(l);
    return 0;
}

int print_VERTEX_T(VertexType* v){
    if(v==NULL)
        return 0;
    print_v_attr(v);
    return 0;
}

int print_EDGE_T(EdgeType* e){
    if(e==NULL)
        return 0;
    print_e_attr(e);
    return 0;
}

int print_GRAPH_T(GraphType* g){
    if(g==NULL)
        return 0;
    print_g(g);
    return 0;
}

int print_attr(Attribute* attr){
    if(attr==NULL)
        die(-1, "print_attr: <null> attribute \n");
    switch(attr->type){
        case BOOL_T:
            printf((attr->value.bv)? "TRUE" : "FALSE" );
            break;
        case INT_T:
            printf("%d", attr->value.iv);
            break;
        case FLOAT_T:
            printf("%f", attr->value.fv);
            break;
        case STRING_T:
            printf("%s", (attr->value.sv)->str);
            break;
        default:
            die(-1, "print_attr: unsupported type\n");
            break;
    }
    return 0;
}

void die(int lno, char* fmt, ...){
    va_list args;
    va_start(args, fmt);
    fprintf(stderr, "FATAL ERROR:");
    if(lno<0)
        fprintf(stderr, ": ");
    else
        fprintf(stderr, "%d: ", lno);
    vfprintf(stderr, fmt, args);
    va_end(args);
    exit(EXIT_FAILURE);    // die
    return;
}

```

```

}

static Attribute* relational_operator( Attribute* attr1, Attribute* attr2, int op, int lno) {
    if(attr1==NULL || attr2==NULL)
        die(lno, "NULL pointer for attr1 or attr2\n");
    int type1 = attr1->type, type2 = attr2->type, resulttype;
    Attribute* result;
    float f1, f2;
    if(type1 == INT_T)
        f1 = attr1->value.iv;
    else if(type1 == FLOAT_T)
        f1 = attr1->value.fv;
    if(type2 == INT_T)
        f2 = attr2->value.iv;
    else if(type2 == FLOAT_T)
        f2 = attr2->value.fv;
    if(type1 == INT_T && type2 == FLOAT_T ||
        type1 == FLOAT_T && type2 == FLOAT_T ||
        type1 == FLOAT_T && type2 == INT_T ||
        type1 == INT_T && type2 == INT_T){
    switch(op){
        case OP_GT:
            return result = new_attr_BOOL_T(f1>f2);
        case OP_LT:
            return result = new_attr_BOOL_T(f1<f2);
        case OP_GE:
            return result = new_attr_BOOL_T(f1>=f2);
        case OP_LE:
            return result = new_attr_BOOL_T(f1<=f2);
    }
    }else
        die(lno, "relational_operator: unsupported op %d .\n",op);
}

static Attribute* math_operator( Attribute* attr1, Attribute* attr2, int op, int lno) {
    if(attr1==NULL || attr2==NULL)
        die(lno, "NULL pointer for attr1 or attr2\n");
    int type1 = attr1->type, type2 = attr2->type, resulttype = UNKNOWN_T;
    int ia1 = 0, ia2 = 0;
    float fa1 = 0., fa2 = 0.;
    Attribute* result;
    if(type1 == INT_T && type2 == INT_T) {
        result = new_attr(INT_T, NULL);
        resulttype = INT_T;
        ia1 = attr1->value.iv;
        ia2 = attr2->value.iv;
    }
    else if (type1 == INT_T && type2 == FLOAT_T ||
        type1 == FLOAT_T && type2 == FLOAT_T ||
        type1 == FLOAT_T && type2 == INT_T) {
        result = new_attr(FLOAT_T, NULL);
        resulttype = FLOAT_T;
        if(type1 == INT_T) fa1 = (float) attr1->value.iv;
        else fa1 = attr1->value.fv;
        if(type2 == INT_T) fa2 = (float) attr2->value.iv;
        else fa2 = attr2->value.fv;
    }
    if(resulttype == INT_T || resulttype == FLOAT_T){
        switch (op) {
            case OP_ADD :
                if(resulttype == INT_T)
                    result->value.iv = ia1 + ia2;
                else if (resulttype == FLOAT_T)
                    result->value.fv = fa1 + fa2;
                else
                    die(lno, "math_operator: coding error\n");
                break;
            case OP_SUB :
                if(resulttype == INT_T)
                    result->value.iv = ia1 - ia2;
                else if (resulttype == FLOAT_T)
                    result->value.fv = fa1 - fa2;
                else
                    die(lno, "math_operator: coding error\n");
                break;
            case OP_MUL :
                if(resulttype == INT_T)
                    result->value.iv = ia1 * ia2;
                else if (resulttype == FLOAT_T)
                    result->value.fv = fa1 * fa2;

```

```

        else
            die(lno, "math_operator: coding error\n");
        break;
        case OP_DIV :
            if(resulttype == INT_T)
                result->value.iv = ia1 / ia2;
            else if (resulttype == FLOAT_T)
                result->value.fv = fa1 / fa2;
            else
                die(lno, "math_operator: coding error\n");
        break;
        default:
            die(lno, "math_operator: unsupported op %d.\n",op);
    }
}
else
    die(lno, "math_operator: unsupported op %d.\n",op);
return result;
}

static Attribute* logic_operator( Attribute* attr1, Attribute* attr2, int op, int lno) {
    if(attr1==NULL || attr2==NULL)
        die(lno, "NULL pointer for attr1 or attr2\n");
    int type1 = attr1->type, type2 = attr2->type, resulttype;
    Attribute* result;
    if(attr1->type == BOOL_T && attr2->type == BOOL_T){
        switch(op){
            case OP_AND:
                return new_attr_BOOL_T(attr1->value.bv && attr2->value.bv);
            case OP_OR:
                return new_attr_BOOL_T(attr1->value.bv || attr2->value.bv);
            default:
                die(lno, "logic_operator: unsupported op %d.",op);
        }
    }
    else
        die(lno, "logic_operator: unsupported op %d.",op);
}

static Attribute* equal_operator( Attribute* attr1, Attribute* attr2, int op, int lno) {
    if(attr1==NULL || attr2==NULL)
        die(lno, "NULL pointer for attr1 or attr2\n");
    int type1 = attr1->type, type2 = attr2->type, resulttype;
    Attribute* result;
    if(type1 == type2){
        switch(type1){
            case INT_T:
                return result = new_attr_BOOL_T( (op==OP_EQ) ? (attr1->value.iv==attr2->value.iv) : (attr1->value.iv !=
                    attr2->value.iv) );
            case FLOAT_T:
                return result = new_attr_BOOL_T( (op==OP_EQ) ? (attr1->value.fv==attr2->value.fv) : (attr1->value.fv !=
                    attr2->value.fv) );
            case BOOL_T:
                return result = new_attr_BOOL_T( (op==OP_EQ) ? (attr1->value.bv==attr2->value.bv) : (attr1->value.bv !=
                    attr2->value.bv) );
            case STRING_T:
                return result = new_attr_BOOL_T( (op==OP_EQ) ? strcmp(attr1->value.sv->str, attr2->value.sv->str)==0 :
                    strcmp(attr1->value.sv->str, attr2->value.sv->str)!=0 );
        }
    }
    else {
        return new_attr_BOOL_T( false );
    }
}

Attribute* binary_operator( Attribute* attr1, Attribute* attr2, int op, int reverse, int rml, int rm2, int lno) {
    if(attr1==NULL || attr2==NULL)
        die(lno, "NULL pointer for attr1 or attr2\n");
    if(reverse) {
        Attribute* tmp = attr1;
        attr1 = attr2;
        attr2 = tmp;
    }
    Attribute* result;
    switch (op) {
        case OP_ADD:
        case OP_SUB:
        case OP_MUL:
        case OP_DIV:
            result = math_operator(attr1, attr2, op, lno);break;
        case OP_GT:

```



```

        case OP_LT:
        case OP_GE:
        case OP_LE:
            result = relational_operator(attr1, attr2, op, lno); break;
        case OP_AND:
        case OP_OR:
            result = logic_operator(attr1, attr2, op, lno); break;
        case OP_EQ:
        case OP_NE:
            result = equal_operator(attr1, attr2, op, lno); break;
        default:
            die(lno, "binary_opertor: unsupported OP %d.\n", op);
    }
    if(rml==FLAG_DESTROY_ATTR) destroy_attr(attr1);
    if(rm2==FLAG_DESTROY_ATTR) destroy_attr(attr2);

    return result;
}

//static = attr1
void assign_operator_to_static(Attribute* attr1, int type, void* value, int rm_attr1, int lno){
    if(attr1 == NULL)
        die(lno, "assign_operator_to_static: <null> Attribute error\n ");
    int type1 = attr1->type;
    if(type1 == type){
        switch(type){
            case INT_T:
                *(int*)value = attr1->value.iv;
                break;
            case FLOAT_T:
                *(float*)value = attr1->value.fv;
                break;
            case BOOL_T:
                *(bool*)value = attr1->value.bv;
                break;
            case STRING_T:
                value = attr1->value.sv;
                break;
            default:
                die(lno, "assign_operator_to_static: incompatible type\n");
        }
    }
    else if(type1==FLOAT_T && type==INT_T){
        *(int*)value = (int)(attr1->value.fv);
    }
    else if(type1==INT_T && type==FLOAT_T){
        *(float*)value = (float)(attr1->value.iv);
    }
    else
        die(lno, "assign_operator_to_static: incompatible type\n");

    if(rm_attr1==FLAG_DESTROY_ATTR) destroy_attr(attr1);
}

//attr1 = attr2
Attribute* assign_operator(Attribute* attr1, Attribute* attr2, int rm_attr1, int rm_attr2, int lno){
    if(attr1 == NULL || attr2 == NULL)
        die(lno, "assign_operator: <null> Attribute error\n");
    if(attr1->type == UNKNOWN_T) attr1->type = attr2->type;
    int type1 = attr1->type, type2 = attr2->type;
    if(type1 == type2){
        switch(type1){
            case INT_T:
                attr1->value.iv = attr2->value.iv;
                break;
            case FLOAT_T:
                attr1->value.fv = attr2->value.fv;
                break;
            case BOOL_T:
                attr1->value.bv = attr2->value.bv;
                break;
            case STRING_T:
                if(attr1->value.sv != NULL) g_string_free(attr1->value.sv,1);
                attr1->value.sv = g_string_new( (attr2->value.sv)->str );
                break;
            default:
                die( lno, "assign_operator : incompatible type.\n");
        }
    }
    else if(type1==FLOAT_T && type2==INT_T){

```

```

    attr1->value.fv = (float)(attr1->value.iv);
}
else if(type1==INT_T && type2==FLOAT_T){
    attr1->value.iv = (int)(attr2->value.fv);
}
else
    die( lno, "assign_operator : incompatible type.\n");

if(rm_attr2==FLAG_DESTROY_ATTR) destroy_attr(attr2);

return attr1;
}

Attribute* unary_operator(Attribute* attr1, int op,int rm_attr1, int lno){
    if(attr1 == NULL)
        die(lno, "NULL Attribute error");
    int type1 = attr1->type;
    Attribute* result;
    switch(op){
        case OP_PLUS:
            if(type1 == INT_T){
                result = new_attr(INT_T, NULL);
                result->value.iv = +(attr1->value.iv);
            }
            else if(type1 == FLOAT_T){
                result = new_attr(FLOAT_T, NULL);
                result->value.fv = +(attr1->value.fv);
            }
            else
                die(lno, "unary_operator: incompatible type.\n");
            break;
        case OP_MINUS:
            if(type1 == INT_T){
                result = new_attr(INT_T, NULL);
                result->value.iv = -(attr1->value.iv);
            }
            else if(type1 == FLOAT_T){
                result = new_attr(FLOAT_T, NULL);
                result->value.fv = -(attr1->value.fv);
            }
            else
                die(lno, "unary_operator: incompatible type.\n");
            break;
        case OP_NOT:
            if(type1 == BOOL_T){
                result = new_attr(BOOL_T, NULL);
                result->value.bv = !(attr1->value.bv);
            }
            else
                die(lno, "unary_operator: incompatible type %d.\n", type1);
            break;
        default:
            die(lno, "unary_operator: unknown operator.\n");
            break;
    }
    if(rm_attr1==FLAG_DESTROY_ATTR) destroy_attr(attr1);
    return result;
}

Attribute* cast_operator(Attribute* attr1, int type, int rm_attr1, int lno){
    if(attr1 == NULL)
        die(lno, "NULL Attribute error");
    int type1 = attr1->type;
    Attribute* result;
    if(type1 == INT_T && type == FLOAT_T){
        result = new_attr(FLOAT_T, NULL);
        result->value.fv = (float)attr1->value.iv;
    }
    else if(type1 == FLOAT_T && type == INT_T){
        result = new_attr(INT_T, NULL);
        result->value.iv = (int)attr1->value.fv;
    }
    else if(type1 == INT_T && type == INT_T){
        result = new_attr(INT_T, NULL);
        result->value.iv = attr1->value.iv;
    }
    else if(type1 == FLOAT_T && type == FLOAT_T){
        result = new_attr(FLOAT_T, NULL);
        result->value.fv = attr1->value.fv;
    }
    else

```

```

        die(lno, "cast_operator: illegal type conversion ");

    if(rm_attrl==FLAG_DESTROY_ATTR) destroy_attr(attrl);
    return result;
}

Attribute* object_get_attribute(void* v, int obj, char* attribute, int autoNew, int lno){
    if(v==NULL)
        die(lno, "object_get_attribute: null object\n");
    Attribute* attr = NULL;
    switch(obj){
        case VERTEX_T:
            attr = vertex_get_attribute((VertexType*)v, attribute, autoNew, lno);
            break;
        case EDGE_T:
            attr = edge_get_attribute((EdgeType*)v, attribute, autoNew, lno);
            break;
        default:
            die(lno, "object_get_attribute: illegal object type\n");
    }
    if(attr==NULL)
        die(lno, "object_get_attribute: attribute '%s' not exist.\n");
    return attr;
}

ListType* list_match(ListType* l, bool (*func) (void*, int), int rm_l){
    if(l==NULL)
        die(-1, "NULL parameter error \n");
    ListType* newl = (ListType*)malloc(sizeof(ListType));
    newl->type = l->type;
    int length = g_list_length(l->list);
    int i, b;
    void* obj;
    for(i=0; i<length; i++){
        obj = g_list_nth_data(l->list, i);
        switch(l->type){
            case VERTEX_T:
                b = func(obj, VERTEX_T);
                break;
            case EDGE_T:
                b = func(obj, EDGE_T);
                break;
            default:
                die(-1, "Illegal type error \n");
        }
        if(b){
            newl->list = g_list_append(newl->list, obj);
        }
    }
    if(rm_l == FLAG_DESTROY_ATTR) destroy_list(l);
    return newl;
}

ListType* list_pipe(ListType* l, int type, int pipe_op, int rm_l){
    ListType* newl = (ListType*)malloc(sizeof(ListType));
    newl->list = NULL;
    newl->type = type;
    int len = g_list_length(l->list);
    int i;
    for(i=0; i<len; i++){
        switch(type){
            case EDGE_T:
                if(pipe_op==OP_OUT)
                    newl = list_append_gl(newl, ((VertexType*)g_list_nth_data(l->list, i))->outEdges, EDGE_T);
                else if(pipe_op==OP_IN)
                    newl = list_append_gl(newl, ((VertexType*)g_list_nth_data(l->list, i))->inEdges, EDGE_T);
                else
                    die(-1, "illegal pipe op for vlist\n");
                break;
            case VERTEX_T:
                if(pipe_op==OP_SV)
                    newl = list_append(newl, VERTEX_T, ((EdgeType*)g_list_nth_data(l->list, i))->start);
                else if(pipe_op==OP_EV)
                    newl = list_append(newl, VERTEX_T, ((EdgeType*)g_list_nth_data(l->list, i))->end);
                else
                    die(-1, "illegal pipe op for elist\n");
                break;
            default: die(-1, "illegal pipe type \n");
        }
    }
}

```

```

    }
    if (rm_l == FLAG_DESTROY_ATTR) destroy_list(l);
    return newl;
}

ListType*
assign_operator_list(ListType** l1, ListType* l2) {
    if (*l1 != NULL) destroy_list(*l1); //gcDef(*l1, LIST_T);
    gcRef(l2, LIST_T);
    return (*l1 = l2);
}

VertexType*
assign_operator_vertex(VertexType** v1, VertexType* v2) {
    if (*v1 != NULL) destroy_vertex(*v1); //gcDef(*v1, VERTEX_T);
    gcRef(v2, VERTEX_T);
    return (*v1 = v2);
}

EdgeType*
assign_operator_edge(EdgeType** e1, EdgeType* e2) {
    if (*e1 != NULL) destroy_edge(*e1); //gcDef(*e1, EDGE_T);
    gcRef(e2, EDGE_T);
    return (*e1 = e2);
}

GraphType*
assign_operator_graph(GraphType** g1, GraphType* g2) {
    if (*g1 != NULL) destroy_graph(*g1); //gcDef(*g1, GRAPH_T);
    gcRef(g2, GRAPH_T);
    return (*g1 = g2);
}

Attribute*
assign_operator_attr(Attribute** a1, Attribute* a2) {
    if (*a1 != NULL) destroy_attr(*a1); //gcDef(*a1, DYN_ATTR_T);
    gcRef(a2, DYN_ATTR_T);
    return *a1 = a2;
}

StringType* assign_operator_string(StringType** s1, StringType* s2) {
    if (*s1 != NULL) destroy_string(*s1);
    gcRef(s2, STRING_T);
    return (*s1 = s2);
}

```

../src/Derivedtype.c

```

// Author : Lixing
#include "Derivedtype.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
int main(int argc, char** argv){
    GraphType* g1 = new_graph();
    new_vertex();
    new_edge();
    //create node for Jack
    VertexType* v1 = new_vertex();
    char* v1_name = "Jack";
    int v1_age = 28;
    float v1_weight = 60.5;
    vertex_assign_attribute(v1, "name", v1_name, STRING_T);
    vertex_assign_attribute(v1, "age", &v1_age, INT_T);
    vertex_assign_attribute(v1, "weight", &v1_weight, FLOAT_T);
    char* tss = "namechanged";
    vertex_assign_attribute(v1, "name", tss, STRING_T);
    print_v(v1);
    print_v_attr(v1);

    //create node for Tom
    VertexType* v2 = new_vertex();
    char* v2_name = "Tom";
    int v2_age = 29;
    float v2_weight = 80.9;
    vertex_assign_attribute(v2, "name", v2_name, STRING_T);
    vertex_assign_attribute(v2, "age", &v2_age, INT_T);
    vertex_assign_attribute(v2, "weight", &v2_weight, FLOAT_T);
    print_v(v2);
    print_v_attr(v2);

    //create node for Jim
    VertexType* v3 = new_vertex();
    char* v3_name = "Jim";
}

```

```

int v3_age = 26;
float v3_weight = 62.5;
vertex_assign_attribute(v3, "name", v3_name, STRING_T);
vertex_assign_attribute(v3, "age", &v3_age, INT_T);
vertex_assign_attribute(v3, "weight", &v3_weight, FLOAT_T);
print_v(v3);
print_v_attr(v3);

//create node for Kate
VertexType* v4 = new_vertex();
char* v4_name = "Kate";
int v4_age = 24;
float v4_weight = 54.5;
vertex_assign_attribute(v4, "name", v4_name, STRING_T);
vertex_assign_attribute(v4, "age", &v4_age, INT_T);
vertex_assign_attribute(v4, "weight", &v4_weight, FLOAT_T);
print_v(v4);
print_v_attr(v4);

//create node for Lily
VertexType* v5 = new_vertex();
char* v5_name = "Lily";
int v5_age = 25;
float v5_weight = 52.5;
vertex_assign_attribute(v5, "name", v5_name, STRING_T);
vertex_assign_attribute(v5, "age", &v5_age, INT_T);
vertex_assign_attribute(v5, "weight", &v5_weight, FLOAT_T);
print_v(v5);
print_v_attr(v5);

//create node for Sarah
VertexType* v6 = new_vertex();
char* v6_name = "Sarah";
int v6_age = 25;
float v6_weight = 55.5;
vertex_assign_attribute(v6, "name", v6_name, STRING_T);
vertex_assign_attribute(v6, "age", &v6_age, INT_T);
vertex_assign_attribute(v6, "weight", &v6_weight, FLOAT_T);
print_v(v6);
print_v_attr(v6);

//create node for John
VertexType* v7 = new_vertex();
char* v7_name = "John";
int v7_age = 29;
float v7_weight = 64.5;
vertex_assign_attribute(v7, "name", v7_name, STRING_T);
vertex_assign_attribute(v7, "age", &v7_age, INT_T);
vertex_assign_attribute(v7, "weight", &v7_weight, FLOAT_T);
print_v(v7);
print_v_attr(v7);

//create node for Jerry
VertexType* v8 = new_vertex();
char* v8_name = "Jerry";
int v8_age = 26;
float v8_weight = 61.5;
vertex_assign_attribute(v8, "name", v8_name, STRING_T);
vertex_assign_attribute(v8, "age", &v8_age, INT_T);
vertex_assign_attribute(v8, "weight", &v8_weight, FLOAT_T);
print_v(v8);
print_v_attr(v8);

//create node for Jack -> Tom
EdgeType* e1 = new_edge();
edge_assign_direction(e1, v1, v2);
char* e1_relation = "friends";
edge_assign_attribute(e1, "relation", e1_relation, STRING_T);
print_e(e1);
print_e_attr(e1);

//create node for Jim -> Tom
EdgeType* e2 = new_edge();
edge_assign_direction(e2, v3, v2);
char* e2_relation = "friends";
edge_assign_attribute(e2, "relation", e2_relation, STRING_T);
print_e(e2);
print_e_attr(e2);

//create node for Tom -> Lily
EdgeType* e3 = new_edge();

```

```

edge_assign_direction(e3, v2, v5);
char* e3_relation = "friends";
edge_assign_attribute(e3, "relation", e3_relation, STRING_T);
print_e(e3);
print_e_attr(e3);

//create node for Jim -> Sarah
EdgeType* e4 = new_edge();
edge_assign_direction(e4, v3, v6);
char* e4_relation = "brother_sister";
edge_assign_attribute(e4, "relation", e4_relation, STRING_T);
print_e(e4);
print_e_attr(e4);

//create node for Jim -> Kate
EdgeType* e5 = new_edge();
edge_assign_direction(e5, v3, v4);
char* e5_relation = "friends";
edge_assign_attribute(e5, "relation", e5_relation, STRING_T);
print_e(e5);
print_e_attr(e5);

//create node for Kate -> Lily
EdgeType* e6 = new_edge();
edge_assign_direction(e6, v4, v5);
char* e6_relation = "sisters";
edge_assign_attribute(e6, "relation", e6_relation, STRING_T);
print_e(e6);
print_e_attr(e6);

//create node for Lily -> Jack
EdgeType* e7 = new_edge();
edge_assign_direction(e7, v5, v1);
char* e7_relation = "friends";
edge_assign_attribute(e7, "relation", e7_relation, STRING_T);
print_e(e7);
print_e_attr(e7);

//create node for Jack -> John
EdgeType* e8 = new_edge();
edge_assign_direction(e8, v1, v7);
char* e8_relation = "friends";
edge_assign_attribute(e8, "relation", e8_relation, STRING_T);
print_e(e8);
print_e_attr(e8);

//create node for Jerry -> Jack
EdgeType* e9 = new_edge();
edge_assign_direction(e9, v8, v1);
char* e9_relation = "brothers";
edge_assign_attribute(e9, "relation", e9_relation, STRING_T);
print_e(e9);
print_e_attr(e9);

//create node for Lily -> Sarah
EdgeType* e10 = new_edge();
edge_assign_direction(e10, v7, v6);
char* e10_relation = "friends";
edge_assign_attribute(e10, "relation", e10_relation, STRING_T);
print_e(e10);
print_e_attr(e10);

//create node for John -> Jerry
EdgeType* e11 = new_edge();
edge_assign_direction(e11, v7, v8);
char* e11_relation = "friends";
edge_assign_attribute(e11, "relation", e11_relation, STRING_T);
print_e(e11);
print_e_attr(e11);

//create node for Jerry -> Sarah
EdgeType* e12 = new_edge();
edge_assign_direction(e12, v8, v6);
char* e12_relation = "siblings";
edge_assign_attribute(e12, "relation", e12_relation, STRING_T);
print_e(e12);
print_e_attr(e12);

g_insert_v(g1, v1);
g_insert_v(g1, v2);
g_insert_v(g1, v3);

```

```

g_insert_v(g1, v4);
g_insert_v(g1, v5);
g_insert_v(g1, v6);
g_insert_v(g1, v7);
g_insert_v(g1, v8);

g_insert_e(g1, e1);
g_insert_e(g1, e2);
g_insert_e(g1, e3);
g_insert_e(g1, e4);
g_insert_e(g1, e5);
g_insert_e(g1, e6);
g_insert_e(g1, e7);
g_insert_e(g1, e8);
g_insert_e(g1, e9);
g_insert_e(g1, e10);
g_insert_e(g1, e11);
g_insert_e(g1, e12);

ListType* lv = list_declaration(VERTEX_T,3, v1, v2, v3);
print_list(lv);
list_append(lv, VERTEX_T, v4);
print_list(lv);
list_append(lv, EDGE_T, e1);
print_list(lv);
list_assign_element(lv, VERTEX_T, 2, v6);
print_list(lv);
ListType* le = list_declaration(-1, 0);
print_list(le);
list_append(le, EDGE_T, e1);
print_list(le);
list_assign_element(le, EDGE_T, 4, e2);
print_list(le);

Attribute* ba = binary_operator(vertex_get_attribute(v1, "age"), vertex_get_attribute(v2, "age"), OP_ADD, 0, 0, 0, 0);
printf("binary operator: %d\n", ba->value.iv);

int s_int;
assign_operator_to_static(ba, INT_T, &s_int, 0, 0);
printf("assignt_static: %d\n", s_int);

Attribute* ap = assign_operator(vertex_get_attribute(v1, "age"), vertex_get_attribute(v2, "age"), 0, 0, 0);
printf("assign: %d\n", ap->value.iv);

Attribute* upa = unary_operator(ba, OP_MINUS, 0, 0);
printf("unary: %d\n", upa->value.iv);

Attribute* ca = cast_operator(ba, FLOAT_T, 0, 0);
printf("cast: %f\n", ca->value.fv);

print_g(g1);
destroy_edge(e1);
print_g(g1);
destroy_vertex(v1);
destroy_vertex(v2);
destroy_vertex(v3);
destroy_vertex(v4);

destroy_vertex(v5);
destroy_vertex(v6);
print_g(g1);
}

```

../src/testonly/graph_lib_test.c

9 NSBL I/O library

```

<graph>

<vertices>
  <vertex>
    <vertex_id></vertex_id>
    <outedges>
      <outedgeID></outedgeID>

```

```

    </outedges>
    <inedges>
      <inedgeID></inedgeID>
    </inedges>
    <vertex_attributes>
      <vertex_attribute>
        <vertex_attribute_name></vertex_attribute_name>
        <vertex_attribute_value></vertex_attribute_value>
        <vertex_attribute_val_type></vertex_attribute_val_type>
      </vertex_attribute>
    </vertex_attributes>
  </vertex>
</vertices>

<edges>
  <edge>
    <edge_id></edge_id>
    <startVID></startVID>
    <endVID></endVID>
    <edge_attributes>
      <edge_attribute>
        <edge_attribute_name></edge_attribute_name>
        <edge_attribute_value></edge_attribute_value>
        <edge_attribute_val_type></edge_attribute_val_type>
      </edge_attribute>
    </edge_attributes>
  </edge>
</edges>
</graph>

```

../src/graph.xml

```

// author : Chantal, Kunal

#ifndef NSBLIO_H_NSBL
#define NSBLIO_H_NSBL

#include <stdio.h>
#include <string.h>
#include "Derivedtype.h"

//print
void print_INT_T(int val);
void print_FLOAT_T(float val);
void print_STRING_T(GString *s);
void print_BOOL_T(bool b);
void print();
void print_NEWLINE();

#endif

```

../src/NSBLio.h

```

// author : Chantal, Kunal

#include <string.h>
#include <stdlib.h>
#include "NSBLio.h"

//function to print string
void print_STRING_T(GString* s)
{
  if(s==NULL) return;
  printf("%s",s->str);
}

//function to print float
void print_FLOAT_T(float val)
{
  printf("%f",val);
}

//function to print integer
void print_INT_T(int val)
{
  printf("%d",val);
}

void print_BOOL_T(bool b)

```



```

{
    printf( (b) ? "true" : "false" );
}

//void print(){}

//function to break line(newline)
void print_NEWLINE()
{
    printf("\n");
}

```

../src/NSBLio.c

```

// author : Chantal, Kunal
#ifndef FILEREADWRITE_H_NSBL
#define FILEREADWRITE_H_NSBL

#include "../mxmldir/mxml.h"
#include "../mxmldir/config.h"
#include "Derivedtype.h"
#include <stdio.h>
#include <string.h>

//function to save a graph in XML format on disk
void saveGraph(GraphType* g, char* fileloc);

//function to read a graph in XML format and convert it to GraphType
GraphType* readGraph(char* fileLoc);

#endif

```

../src/FileReadWrite.h

```

// author : Chantal, Kunal
#include "FileReadWrite.h"

//Function to write graph in xml format
void saveGraph(GraphType* g, char* fileloc)
{
    FILE *fp; /*File to write*/
    char *delim="/";
    char filepath[100];
    char str[100];

    mxml_node_t *xml;
    mxml_node_t *graph;
    mxml_node_t *graph_id;
    mxml_node_t *vertices_list;
    mxml_node_t *edges_list;
    mxml_node_t *vertices;
    mxml_node_t *vertex;
    mxml_node_t *vertex_id;
    mxml_node_t *outedges;
    mxml_node_t *outedge;
    mxml_node_t *inedges;
    mxml_node_t *inedge;
    mxml_node_t *vertex_attributes;
    mxml_node_t *vertex_attribute;
    mxml_node_t *vertex_attribute_name;
    mxml_node_t *vertex_attribute_value;
    mxml_node_t *vertex_attribute_value_type;
    mxml_node_t *edges;
    mxml_node_t *edge;
    mxml_node_t *edge_id;
    mxml_node_t *startV;
    mxml_node_t *endV;
    mxml_node_t *edge_attributes;
    mxml_node_t *edge_attribute;
    mxml_node_t *edge_attribute_name;
    mxml_node_t *edge_attribute_value;
    mxml_node_t *edge_attribute_value_type;

    xml = mxmlNewXML("1.0");
    graph = mxmlNewElement(xml, "graph");
    //graph_id = mxmlNewElement(graph, "graph_id");
    //mxmlNewText(graph_id, 1, g->id);
    //vertices_list = mxmlNewElement(graph, "vertices_list");

```

```

//mxmNewText(vertices_list, 1, g->vertexIdList);
//edges_list = mxmNewElement(graph, "edges_list");
//mxmNewText(edges_list, 1, g->edgeIdList);

vertices = mxmNewElement(graph, "vertices");
GList* listV= g_hash_table_get_values(g->vertices);
GList* listE= g_hash_table_get_values(g->edges) ;
int lv = g_list_length(g->vertexIdList);
int n = 0;

for(n; n<lv; n++)
{
    VertexType* v = (VertexType*)(g_list_nth_data(listV,n));
    vertex = mxmNewElement(vertices, "vertex");
    vertex_id=mxmNewElement(vertex,"vertex_id");
    int len = snprintf(str, 100, "%d",v->id );
    //printf("%s\n",str);
    mxmNewText(vertex_id,0,str);

    outedges = mxmNewElement(vertex,"outedges");
    int loe= g_list_length(v->outEdges);
    int y=0;
    for (y;y<loe;y++)
    {
        outedge=mxmNewElement(outedges,"outedge");
        EdgeType* e_temp = g_list_nth_data(v->outEdges,y);
        int len = snprintf(str, 100, "%d",e_temp->id );
        mxmNewText(outedge, 0, str);
    }

    inedges = mxmNewElement(vertex,"inedges");

    int lie= g_list_length(v->inEdges);
    int d=0;
    for (d;d<lie;d++)
    {
        inedge=mxmNewElement(inedges,"inedge");
        EdgeType* e_temp = g_list_nth_data(v->inEdges,d);
        int len = snprintf(str, 100, "%d",e_temp->id );
        mxmNewText(inedge, 0, str);
    }

    vertex_attributes = mxmNewElement(vertex,"vertex_attributes");
    //vertex_attribute = mxmNewElement(vertex_attributes,"vertex_attribute");
    GList* v_attr=g_hash_table_get_keys(v->attributes);
    GList* v_attr_value=g_hash_table_get_values(v->attributes);
    int a=g_list_length(v_attr);
    int x=0;

    for(x;x<a;x++)
    {
        vertex_attribute = mxmNewElement(vertex_attributes,"vertex_attribute");
        vertex_attribute_name=mxmNewElement(vertex_attribute,"vertex_attribute_name");
        mxmNewText(vertex_attribute_name, 0,(char *)g_list_nth_data(v_attr,x));
        vertex_attribute_value=mxmNewElement(vertex_attribute, "vertex_attribute_value");
        Attribute* a=(Attribute*)(g_list_nth_data(v_attr_value,x));
        int type=(int )a->type;
        if (type==INT_T)
        {
            len = snprintf(str, 100, "%d",((Attribute*)(g_list_nth_data(v_attr_value,x)))->value.iv);
            mxmNewText(vertex_attribute_value,0,str);
        }
        if (type==FLOAT_T)
        {
            len = snprintf(str, 100, "%f",((Attribute*)(g_list_nth_data(v_attr_value,x)))->value.fv);
            mxmNewText(vertex_attribute_value,0,str);
        }
        if (type==STRING_T)
        {
            mxmNewText(vertex_attribute_value, 0,
                ((Attribute*)(g_list_nth_data(v_attr_value,x)))->value.sv->str);
            //mxmNewText(edge_attribute_value,1, (char*)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value));
        }
        if (type==BOOL_T)
        {
            char* val = ((Attribute*)(g_list_nth_data(v_attr_value,x)))->value.bv ? "true" : "false";
            mxmNewText(vertex_attribute_value, 0, val);
        }

        //printf("%s\n", (char*)((Attribute*)(g_list_nth_data(v_attr_value,x)))->value));
    }
}

```

```

/*
Attribute* attr = (Attribute*) malloc(sizeof(Attribute));
attr = (Attribute*)g_list_nth_data(v_attr_value,x);
long int abc = attr->type;
void *t = attr->value;
fprintf(stderr,"\n\n%d\n\n%d\n\n", abc,sizeof((char *)t));
char* temp=(char*)g_list_nth_data(v_attr,x);
void* t=(vertex_get_attribute_value(v, "Name"));
*/

//char *t1 = (char*)(attr->value);
//char *temp=(char*)((Attribute*)(g_list_nth_data(v_attr_value,x))->value);
//char *templ = (char *)malloc(sizeof(char *)* sizeof(temp));
//printf("%s\n",temp);
//int len = snprintf(str, 100, "%s",temp );
//printf("%s %d\n",temp, sizeof(temp));
//memcpy(templ, temp,sizeof(temp));
// mxmNewText(vertex_attribute_value, 1,
// (char*)((Attribute*)(g_list_nth_data(v_attr_value,x))->value));
//mxmNewText(vertex_attribute_value, 1, templ);
//mxmNewText(vertex_attribute_value, 1, (char*)((Attribute*)(g_list_nth_data(v_attr_value,x))->value));
//mxmNewText(vertex_attribute_value, 1, (char*)(vertex_get_attribute_value(v, g_list_nth_data(v_attr,x))));//
// need to check if cast works as returns void*

vertex_attribute_value_type=mxmNewElement(vertex_attribute, "vertex_attribute_value_type");
//mxmNewText(vertex_attribute_value_type,1,
int len = snprintf(str, 100, "%d",((int)((Attribute*)(g_list_nth_data(v_attr_value,x))->type));
//printf("\nPrinting type of value in vertex by jing :: %s\n", str);
//printf("\nVertex attr value derived by me : %d\n", type);
//printf("\n INT value: %d\n", INT_T);
//printf("\n FLOAT value: %d\n", FLOAT_T);
//printf("\n STRING value: %d\n", STRING_T);
mxmNewText(vertex_attribute_value_type, 0, str);

}
}

edges = mxmNewElement(graph, "edges");
int le = g_list_length(g->edgeIdList);
int m=0;
for(m; m<le; m++)
{
EdgeType* e= g_list_nth_data(listE,m);
edge =mxmNewElement(edges,"edge");
edge_id=mxmNewElement(edge, "edge_id");
int len = snprintf(str, 100, "%d", (e->id) );
mxmNewText(edge_id,0,str);
startV = mxmNewElement(edge, "startV");
len = snprintf(str, 100, "%d", e->start->id );
mxmNewText(startV,0,str);
endV = mxmNewElement(edge,"endV");

len = snprintf(str, 100, "%d", (e->end->id) );
mxmNewText(endV,0,str);
edge_attributes = mxmNewElement(edge, "edge_attributes");
//edge_attribute = mxmNewElement(edge_attributes, "edge_attribute");

GList* e_attr= g_hash_table_get_keys(e->attributes);
GList* e_attr_value=g_hash_table_get_values(e->attributes);
int b=g_list_length(e_attr);
//printf("Just before seg fault1\n");
int c=g_list_length(e_attr_value);
//printf("just before seg fault\n");
//printf("lengths %d %d\n",b,c);
int y=0;
for (y; y<b;y++)
{
edge_attribute = mxmNewElement(edge_attributes, "edge_attribute");
edge_attribute_name=mxmNewElement(edge_attribute,"edge_attribute_name");
mxmNewText(edge_attribute_name,0, (char*)g_list_nth_data(e_attr,y));
edge_attribute_value=mxmNewElement(edge_attribute,"edge_attribute_value");
//mxmNewText(edge_attribute_value, 1,(char*)((Attribute*)(g_list_nth_data(e_attr_value,y))->value));
Attribute* a=(Attribute*)(g_list_nth_data(e_attr_value,y));
int type=(int )a->type;
if (type==INT_T)
{
len = snprintf(str, 100, "%d",((Attribute*)(g_list_nth_data(e_attr_value,y))->value.iv);
mxmNewText(edge_attribute_value,0,str);

```

```

    }
    if (type==FLOAT_T)
    {
        len = snprintf(str, 100, "%f",((Attribute*)(g_list_nth_data(e_attr_value,y)))->value.fv);
        mxmlNewText(edge_attribute_value,0,str);
    }
    if (type==STRING_T)
    {
        mxmlNewText(edge_attribute_value,0, ((Attribute*)(g_list_nth_data(e_attr_value,y)))->value.sv->str);
    }
    if (type==BOOL_T)
    {
        char* val = ((Attribute*)(g_list_nth_data(e_attr_value,y)))->value.bv ? "true" : "false";
        mxmlNewText(edge_attribute_value, 0, val);
    }

    //mxmlNewText(edge_attribute_value, 1,
    //((char*)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value));

    //int len = snprintf(str, 100, "%d", (int)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value);
    //mxmlNewText(edge_attribute_value, 1, (char*)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value);
    //Attribute* a=g_list_nth_data(e_attr_value,y);
    //printf("edge attribute value: %s\n",a->value);
    //((char*)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value));
    //mxmlNewText(edge_attribute_value,1, (char*)((Attribute*)(g_list_nth_data(e_attr_value,y)))->value));
    //need to check if cast works as returns void*
    edge_attribute_value_type=mxmlNewElement(edge_attribute, "edge_attribute_value_type");
    len = snprintf(str, 100, "%d",((int)((Attribute*)(g_list_nth_data(e_attr_value,y)))->type));
    //printf("\nPrinting the attribute value of edge derived from jing: %s\n", str);
    //printf("\nEdge attr value derived by me: %d\n", type);
    mxmlNewText(edge_attribute_value_type, 0, str);

    }
}
/*
strcpy(filepath, fileloc);

printf("%s\n", filepath );
B

if((strcmp(fileloc,delim)!=0)&&(strcmp(fileloc,"./")!=0))
{
    strcat(filepath, delim);
    printf("%s\n", filepath );
}

strcat(filepath, filename);
printf("%s\n", filepath );
*/
fp = fopen(fileloc, "w");
mxmlSaveFile(xml, fp, MXML_NO_CALLBACK);
fclose(fp);
//printf("%s xml file written\n\n",fileloc);

}

//Function to read a saved xml graph
GraphType* readGraph(char* fileloc)
{
    FILE *fp;
    mxml_node_t *tree;
    mxml_node_t *node;
    mxml_node_t *node_loop;
    mxml_node_t *node_v;
    mxml_node_t *node_temp1;
    mxml_node_t *node_temp2;
    char *delim="/*";
    char *temp_eID;
    char *temp_vID;
    GraphType* g;
    VertexType* v;
    EdgeType* e;
    GList* edge_list;
    GList* vertices_list;
    GList* check_edgesID;
    GList* check_verticesID;
    char temp[100];
    char str[100];
    /*

```

```

strcpy(filepath, fileloc);

printf("%s\n", filepath );

if((strcmp(fileloc,delim)!=0)&&(strcmp(fileloc,"./")!=0))
{
    strcat(filepath, delim);
    printf("%s\n", filepath );
}

strcat(filepath, filename);
printf("%s\n", filepath );
*/

fp = fopen(fileloc, "r");
/*invalid or empty file name check */
if (fp==NULL)
{
    printf("Error in provided file name. Please check file name again.\n");
    return 0;
}
tree = mxmllLoadFile(NULL, fp,MXML_TEXT_CALLBACK);
/*xml file check */
if (tree==NULL)
{
    printf("The file provided is not an XML file.\n");
    return 0;
}
fclose(fp);

//printf("xml loaded\n");

//set new VERTEX IDs
for (node = mxmllFindElement(tree, tree,
                             "vertex",
                             NULL, NULL,
                             MXML_DESCEND);

     node != NULL;
     node = mxmllFindElement(node, tree,
                             "vertex",
                             NULL, NULL,
                             MXML_DESCEND))
{
    node = mxmllGetFirstChild(node); //vertex id
    char *old_vID= (char*)malloc(sizeof(char));
    strcpy(old_vID ,mxmllGetText(node,NULL));
    //printf("old vertex id: %s\n", old_vID);
    int len = snprintf(str, 100, "%ld", (long int)new_vertexId());
    //printf("new vertex id: %s\n", str);
    //char *new_vID=(char *)((long int)new_vertexId());
    mxmllSetText(node, 0, str);
    //printf("vertex id updated\n");

    //updating new vertexID at every startV node
    for (node_loop = mxmllFindElement(tree, tree,
                                       "startV",
                                       NULL, NULL,
                                       MXML_DESCEND);

         node_loop != NULL;
         node_loop = mxmllFindElement(node_loop, tree,
                                       "startV",
                                       NULL, NULL,
                                       MXML_DESCEND))
    {
        //printf("node text: %s\n",mxmllGetText(node_loop,NULL));
        //printf("old vid: %s\n",old_vID);
        //printf("str cmp%d\n", strcmp(old_vID,mxmllGetText(node_loop,NULL)));
        if ((strcmp(old_vID,mxmllGetText(node_loop,NULL))==0))
        {
            mxmllSetText(node_loop, 0, str);
            //printf("updated startV\n");
        }
        //else
        //printf("didnt update startV\n");
    }

    //updating new vertexID at every endV node

```

```

for (node_loop = mxmlFindElement(tree, tree,
    "endV",
    NULL, NULL,
    MXML_DESCEND);
node_loop != NULL;
node_loop = mxmlFindElement(node_loop, tree,
    "endV",
    NULL, NULL,
    MXML_DESCEND))
{
    if ((strcmp(old_vID, mxmlGetText(node_loop, NULL)) == 0))
    {
        mxmlSetText(node_loop, 0, str);
        //printf("updated endV\n");
    }
    //else
    //printf("didn't update endV\n");
}
}

//set new EDGE IDs
for (node = mxmlFindElement(tree, tree,
    "edge",
    NULL, NULL,
    MXML_DESCEND);
node != NULL;
node = mxmlFindElement(node, tree,
    "edge",
    NULL, NULL,
    MXML_DESCEND))
{
    node = mxmlGetFirstChild(node); //edge id
    char *old_eID = (char *) malloc(sizeof(char));
    strcpy(old_eID, mxmlGetText(node, NULL));
    //printf("old edge id: %s\n", old_eID);
    int len = snprintf(str, 100, "%ld", (long int) new_edgeId());
    //printf("new edge id: %s\n", str);
    //char *new_vID = (char *) ((long int) new_vertexId());
    mxmlSetText(node, 0, str);

    //updating new edgeID at every outedge node
    for (node_loop = mxmlFindElement(tree, tree,
        "outedge",
        NULL, NULL,
        MXML_DESCEND);
node_loop != NULL;
node_loop = mxmlFindElement(node_loop, tree,
    "outedge",
    NULL, NULL,
    MXML_DESCEND))
    {
        //printf("outedge node text: %s\n", mxmlGetText(node_loop, NULL));
        //printf("old eid: %s\n", old_eID);
        //printf("str cmp: %d\n", strcmp(old_eID, mxmlGetText(node_loop, NULL)));

        if ((strcmp(old_eID, mxmlGetText(node_loop, NULL)) == 0))
        {
            mxmlSetText(node_loop, 0, str);
            //printf("updated outedge\n");
        }
        //else
        //printf("not updated outedge\n");
    }

    //updating new edgeID at every inedge node
    for (node_loop = mxmlFindElement(tree, tree,
        "inedge",
        NULL, NULL,
        MXML_DESCEND);
node_loop != NULL;
node_loop = mxmlFindElement(node_loop, tree,
    "inedge",
    NULL, NULL,
    MXML_DESCEND))
    {
        //printf("inedge node text: %s\n", mxmlGetText(node_loop, NULL));
        //printf("old eid: %s\n", old_eID);
    }
}

```

```

//printf("str cmp: %d\n", strcmp(old_eID,mxmlGetText(node_loop,NULL)));

if ((strcmp(old_eID,mxmlGetText(node_loop,NULL))==0))

{
    mxmlSetText(node_loop, 0, str);
    //printf("updated inedge\n");
}
//else
    //printf("not updated inedge\n");
}

}

//creating a new graph so as to copy the old graph into it.
g=new_graph();
//creating vertex from the xml
for (node = mxmlFindElement(tree, tree,
                            "vertex",
                            NULL, NULL,
                            MXML_DESCEND);

     node != NULL;
     node = mxmlFindElement(node, tree,
                            "vertex",
                            NULL, NULL,
                            MXML_DESCEND))

{
    v=new_vertex();
    node_v = mxmlGetFirstChild(node); //vertex id
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_v));
    v->id=atoi(mxmlGetText(node_v,NULL));
    //printf("node vertex text: %s\n",mxmlGetText(node_v,NULL));
    //printf("new vertex id: %ld\n", v->id);
    node_v = mxmlGetNextSibling(node_v); //skip outedges
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_v));
    node_v = mxmlGetNextSibling(node_v); //skip inedges
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_v));
    node_v = mxmlGetNextSibling(node_v); //populate attributes for the vertex
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_v));
    //node_temp1 = mxmlGetLastChild(node_v);
    node_temp1 = mxmlGetFirstChild(node_v);
    // printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_temp1));
    //node_temp2 = mxmlGetFirstChild(node_temp1);
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_temp2));
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetText(node_temp2,NULL));
    //int *v =(int*)malloc(sizeof(int));
    while(node_temp1!=NULL)
    {
        node_temp2 = mxmlGetFirstChild(node_temp1);
        //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_temp2));
        //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetText(node_temp2,NULL));
        char* attribute = (char *) mxmlGetText(node_temp2,NULL); //attr name
        node_temp2=mxmlGetNextSibling(node_temp2); //go to attr value
        mxml_node_t *n=node_temp2;
        //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_temp2));
        void* value; //mxmlGetText(node_temp2,NULL); //needs to be checked
        node_temp2=mxmlGetNextSibling(node_temp2); //go to attr value type
        //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetElement(node_temp2));
        int type=atoi(mxmlGetText(node_temp2,NULL));
        if (type==INT_T)
        {
            int val=atoi(mxmlGetText(n,NULL));
            //printf("printing int val: %d\n",val);
            int *v1 =(int*)malloc(sizeof(int));
            *v1=val;

            vertex_assign_attribute( v, attribute, v1, type);
            //printf("\nattribute assigned to vertex\n");
            //value=&val;
        }
        if (type==FLOAT_T)
        {
            float val=atof(mxmlGetText(n,NULL));
            //printf("printing float val: %f\n",val);
            float* f1=(float*)malloc(sizeof(float));
            *f1=val;
            vertex_assign_attribute( v, attribute, f1, type);
            //printf("\nattribute assigned to vertex\n");
            //value=&val;
        }
        if (type==STRING_T)

```

```

        {
            //value= (char *)mxmGetText(n,NULL);
            char* c = (char*) malloc(sizeof(char));
            c = (char*)mxmGetText(n,NULL);
            //printf("printing char val: %s\n",c);
            GString* s=g_string_new(c);
            vertex_assign_attribute( v, attribute, s, type);
            //printf("\nattribute assigned to edge\n");
        }
    if (type==BOOL_T)
    {
        char * vIn = (char*)mxmGetText(n,NULL);
        bool vOut = vIn && strcasecmp(vIn,"true")==0;
        bool* bl=(bool*)malloc(sizeof(bool));
        *bl = vOut;
        vertex_assign_attribute( v, attribute, bl, type);
    }
    //vertex_assign_attribute( v, attribute, value, type); //check if its correct as it returns int
    //testing if vertex assigned attribute or not
    //printf("vertex attribute value: %d\n",vertex_get_attribute_value(v, attribute));
    node_templ=mxmGetNextSibling(node_templ);
}
//testing if vetrex is assigned attributue or not .
//print_v_attr(v);
//inserting vertex into graph
g_insert_v(g, v); //check if its correct as it returns int
//printf("vertex inserted\n");

}
//creating edge from the xml
for (node = mxmFindElement(tree, tree,
                            "edge",
                            NULL, NULL,
                            MXML_DESCEND);
     node != NULL;
     node = mxmFindElement(node, tree,
                            "edge",
                            NULL, NULL,
                            MXML_DESCEND))
{
    VertexType* sv;
    VertexType* ev;
    e=new_edge();
    node_v = mxmGetFirstChild(node); //edge id
    e->id=atoi(mxmGetText(node_v,NULL));
    //printf("edge id: %d\n", e->id);
    node_v = mxmGetNextSibling(node_v);
    //printf("check node name in edge: %s\n", mxmGetElement(node_v));

    int startVid=atoi(mxmGetText(node_v,NULL));
    //printf("startV id: %d\n",startVid);
    //have to check if the hash table lookup works or not
    GList* listV= g_hash_table_get_values(g->vertices);
    int loe= g_list_length(listV);
    int y=0;
    for (y;y<loe;y++)
    {
        VertexType* v = (VertexType*)(g_list_nth_data(listV,y));
        if (v->id==startVid)
            sv=v;
    }
    //printf("sv->id: %d\n",sv->id);

    //sv=(VertexType*)g_hash_table_lookup(g->vertices,startVid);
    node_v = mxmGetNextSibling(node_v);
    //printf("check node name in edge: %s\n", mxmGetElement(node_v));
    int endVid=atoi(mxmGetText(node_v,NULL));
    //printf("endV id: %d\n",endVid);

    //y=0;
    for (y=0;y<loe;y++)
    {
        VertexType* v = (VertexType*)(g_list_nth_data(listV,y));
        if (v->id==endVid)
            ev=v;
    }
    //printf("ev->id: %d\n",ev->id);

    //have to check if the hash table lookup works or not
    // ev=g_hash_table_lookup(g->vertices,(gconstpointer)(long int)endVid);
}

```



```

//the below code populates outedges and inedges as well
edge_assign_direction(e, sv, ev);//check if its correct as it returns int
//go to edge_attributes
//printf("edge direction assigned\n");
node_v = mxmlGetNextSibling(node_v);
//printf("check node name in edge: %s\n", mxmlGetElement(node_v));
//go to edge_attribute(firstchild)
node_temp1 = mxmlGetFirstChild(node_v);
//printf("check node name in edge: %s\n", mxmlGetElement(node_temp1));

//go to edge_attribute_name/value/type
//node_temp1 = mxmlGetLastChild(node_v);
//printf("check node name in edge node_temp1: %s\n", mxmlGetElement(node_v));

//node_temp1 = mxmlGetFirstChild(node_temp1);
//printf("check node name in edge: %s\n", mxmlGetElement(node_v));

while(node_temp1!=NULL)
{
    node_temp2 = mxmlGetFirstChild(node_temp1);
    //printf("node_temp2: %s\n", mxmlGetElement(node_temp2));
    //printf("checkpoint node temp 2(attr name): %s\n", mxmlGetText(node_temp2,NULL));
    char* attribute=(char *)mxmlGetText(node_temp2,NULL);//attr name
    node_temp2=mxmlGetNextSibling(node_temp2);//go to attr value
    mxml_node_t *n=node_temp2;
    void* value;//=mxmlGetText(node_temp2,NULL);//needs to be checked
    node_temp2=mxmlGetNextSibling(node_temp2);//go to attr value type
    int type=atoi(mxmlGetText(node_temp2,NULL));
    if (type==INT_T)
    {
        int val=atoi(mxmlGetText(n,NULL));
        //printf("printing int val: %d\n", val);
        //printf("printing int val: %d\n",val);
        int *v1 =(int*)malloc(sizeof(int));
        *v1=val;

        edge_assign_attribute( e, attribute, v1, type);
        //printf("\nattribute assigned to edge\n");
        //value=&val;
    }
    if (type==FLOAT_T)
    {
        //int len = snprintf(str, 100, "%f", (mxmlGetText(n,NULL)));
        float val=atof(mxmlGetText(n,NULL));
        //printf("printing float val: %f\n",val);
        float *v1 =(float*)malloc(sizeof(float));
        *v1=val;
        edge_assign_attribute( e, attribute, v1, type);
        //printf("\nattribute assigned to edge\n");

        //value=&val;
    }
    if (type==STRING_T)
    {
        //value=(char *)mxmlGetText(n,NULL);
        char* c = (char*) malloc(sizeof(char));
        c = (char*)mxmlGetText(n,NULL);

        GString* s=g_string_new(c);
        edge_assign_attribute( e, attribute, s, type);
        //printf("\nattribute assigned to edge\n");
    }

    if (type==BOOL_T)
    {
        char * vIn = (char*)mxmlGetText(n,NULL);
        bool vOut = vIn && strcasecmp(vIn,"true")==0;
        bool* b1=(bool*)malloc(sizeof(bool));
        *b1 = vOut;
        edge_assign_attribute( e, attribute, b1, type);
    }

    //edge_assign_attribute( e, attribute, value, type);//check if its correct as it returns int
    //printf("\nattribute assigned to edge\n");
    node_temp1=mxmlGetNextSibling(node_temp1);
}
//inserting vertex into graph
g_insert_e(g, e);//check if its correct as it returns int
//printf("edge insrted\n");
}

//test to see if graph getting changed

```

```

//fp = fopen("tryc.xml", "w");
//mxmmlSaveFile(tree, fp, MXML_NO_CALLBACK);
//fclose(fp);
//printf("test xml written\n");
return g;
}

```

../src/FileReadWrite.c

```

// author : Chantal, Kunal
#include "FileReadWrite.h"
int main()
{
    GraphType* g;
    GraphType* g1;

    VertexType* v1;
    VertexType* v2;
    VertexType* v3;
    VertexType* v4;
    VertexType* v5;
    VertexType* v6;
    VertexType* v7;
    VertexType* v8;
    VertexType* v9;
    VertexType* v10;

    EdgeType* e1;
    EdgeType* e2;
    EdgeType* e3;
    EdgeType* e4;
    EdgeType* e5;
    EdgeType* e6;
    EdgeType* e7;
    EdgeType* e8;
    EdgeType* e9;
    EdgeType* e10;
    EdgeType* e11;
    EdgeType* e12;
    EdgeType* e13;
    EdgeType* e14;
    EdgeType* e15;

    e1=new_edge();
    e2=new_edge();
    e3=new_edge();
    e4=new_edge();
    e5=new_edge();
    e6=new_edge();
    e7=new_edge();
    e8=new_edge();
    e9=new_edge();
    e10=new_edge();
    e11=new_edge();
    e12=new_edge();
    e13=new_edge();
    e14=new_edge();
    e15=new_edge();

    v1=new_vertex();
    v2=new_vertex();
    v3=new_vertex();
    v4=new_vertex();
    v5=new_vertex();
    v6=new_vertex();
    v7=new_vertex();
    v8=new_vertex();
    v9=new_vertex();
    v10=new_vertex();

    g=new_graph();
    int v1_age = 28;
    float v1_weight = 60.5;
    //edge_assign_attribute(EdgeType* e, char* attribute, void* value, int type)
    edge_assign_attribute(e1, "e1_testString", "40", 3);
}

```

```

edge_assign_attribute(e1, "e1_weight_float", &v1_weight, 2);
edge_assign_attribute(e1, "e1_weight_int", &v1_age, 1);
edge_assign_attribute(e2, "e2_testFloat", &v1_weight, 2);
edge_assign_attribute(e2, "e2_weight_float", &v1_weight, 2);
edge_assign_attribute(e2, "e2_weight_int", &v1_age, 1);
edge_assign_attribute(e3, "e3_weight_int", &v1_age, 1);
edge_assign_attribute(e3, "e3_testFloat", &v1_weight, 2);
edge_assign_attribute(e3, "e3_testString", "40", 3);
edge_assign_attribute(e4, "e4_weight_int", &v1_age, 1);
edge_assign_attribute(e4, "e4_testFloat", &v1_weight, 2);
edge_assign_attribute(e4, "e4_testString", "40", 3);
edge_assign_attribute(e5, "e5_weight_int", &v1_age, 1);
edge_assign_attribute(e5, "e5_testFloat", &v1_weight, 2);
edge_assign_attribute(e5, "e5_testString", "40", 3);
edge_assign_attribute(e6, "e6_weight_int", &v1_age, 1);
edge_assign_attribute(e6, "e6_testFloat", &v1_weight, 2);
edge_assign_attribute(e6, "e6_testString", "40", 3);
edge_assign_attribute(e7, "e7_weight_int", &v1_age, 1);
edge_assign_attribute(e7, "e7_testFloat", &v1_weight, 2);
edge_assign_attribute(e7, "e7_testString", "40", 3);
edge_assign_attribute(e8, "e8_weight_int", &v1_age, 1);
edge_assign_attribute(e8, "e8_testFloat", &v1_weight, 2);
edge_assign_attribute(e8, "e8_testString", "40", 3);
edge_assign_attribute(e9, "e9_weight_int", &v1_age, 1);
edge_assign_attribute(e9, "e9_testFloat", &v1_weight, 2);
edge_assign_attribute(e9, "e9_testString", "40", 3);
edge_assign_attribute(e10, "e10_weight_int", &v1_age, 1);
edge_assign_attribute(e10, "e10_testFloat", &v1_weight, 2);
edge_assign_attribute(e10, "e10_testString", "40", 3);
edge_assign_attribute(e11, "e11_weight_int", &v1_age, 1);
edge_assign_attribute(e11, "e11_testFloat", &v1_weight, 2);
edge_assign_attribute(e11, "e11_testString", "40", 3);
edge_assign_attribute(e12, "e12_weight_int", &v1_age, 1);
edge_assign_attribute(e12, "e12_testFloat", &v1_weight, 2);
edge_assign_attribute(e12, "e12_testString", "40", 3);
edge_assign_attribute(e13, "e13_weight_int", &v1_age, 1);
edge_assign_attribute(e13, "e13_testFloat", &v1_weight, 2);
edge_assign_attribute(e13, "e13_testString", "40", 3);
edge_assign_attribute(e14, "e14_weight_int", &v1_age, 1);
edge_assign_attribute(e14, "e14_testFloat", &v1_weight, 2);
edge_assign_attribute(e14, "e14_testString", "40", 3);
edge_assign_attribute(e15, "e15_weight_int", &v1_age, 1);
edge_assign_attribute(e15, "e15_testFloat", &v1_weight, 2);
edge_assign_attribute(e15, "e15_testString", "40", 3);

//vertex_assign_attribute(VertexType* v, char* attribute, void* value, int type)

vertex_assign_attribute(v1, "Name", "v1_vertex", 3);
vertex_assign_attribute(v1, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v1, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v2, "Name", "v2_vertex", 3);
vertex_assign_attribute(v2, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v2, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v3, "Name", "v3_vertex", 3);
vertex_assign_attribute(v3, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v3, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v4, "Name", "v4_vertex", 3);
vertex_assign_attribute(v4, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v4, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v5, "Name", "v5_vertex", 3);
vertex_assign_attribute(v5, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v5, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v6, "Name", "v6_vertex", 3);
vertex_assign_attribute(v6, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v6, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v7, "Name", "v7_vertex", 3);
vertex_assign_attribute(v7, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v7, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v8, "Name", "v8_vertex", 3);
vertex_assign_attribute(v8, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v8, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v9, "Name", "v9_vertex", 3);
vertex_assign_attribute(v9, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v9, "TestFloat", &v1_weight, 2);
vertex_assign_attribute(v10, "Name", "v10_vertex", 3);
vertex_assign_attribute(v10, "Test_Int", &v1_age, 1);
vertex_assign_attribute(v10, "TestFloat", &v1_weight, 2);

//edge_assign_direction(EdgeType* e, VertexType* v1, VertexType* v2)

edge_assign_direction(e1, v1, v2);

```

```

edge_assign_direction(e2,v2, v3);
edge_assign_direction(e3,v3, v4);
edge_assign_direction(e4,v4, v3);
edge_assign_direction(e5,v4, v5);
edge_assign_direction(e6,v5, v6);
edge_assign_direction(e7,v6, v7);
edge_assign_direction(e8,v7, v8);
edge_assign_direction(e9,v8, v9);
edge_assign_direction(e10,v9, v10);
edge_assign_direction(e11,v1, v3);
edge_assign_direction(e12,v10, v3);
edge_assign_direction(e13,v2, v4);
edge_assign_direction(e14,v6, v1);
edge_assign_direction(e15,v7, v5);

//g_insert_v(GraphType* g, VertexType* v)

g_insert_v(g, v1);
g_insert_v(g, v2);
g_insert_v(g, v3);
g_insert_v(g, v4);
g_insert_v(g, v5);
g_insert_v(g, v6);
g_insert_v(g, v7);
g_insert_v(g, v8);
g_insert_v(g, v9);
g_insert_v(g, v10);

//g_insert_e(GraphType* g, EdgeType* e)

g_insert_e(g, e1);
g_insert_e(g, e2);
g_insert_e(g, e3);
g_insert_e(g, e4);
g_insert_e(g, e5);
g_insert_e(g, e6);
g_insert_e(g, e7);
g_insert_e(g, e8);
g_insert_e(g, e9);
g_insert_e(g, e10);
g_insert_e(g, e11);
g_insert_e(g, e12);
g_insert_e(g, e13);
g_insert_e(g, e14);
g_insert_e(g, e15);

//test xml creation
//void saveGraph(GraphType* g, char* fileloc, char* filename)

saveGraph(g, "try.xml");

//test read xml
//GraphType* readGraph(char* fileloc, char* filename)

gl=readGraph("try.xml");
VertexType* v;
saveGraph(gl, "tryd.xml");
GList* l=get_g_all(gl);
int le=g_list_length(l);
int n=0;
for (n;n<le;n++)
{
    printf("+++++++\n");
    v=get_end_vertex(g_list_nth_data(l,n));
    printf("edge end vertex: %d\n",v->id);
    print_v_attr(v);
    v=get_start_vertex(g_list_nth_data(l,n));
    printf("edge start vertex: %d\n",v->id);
    print_e_attr(g_list_nth_data(l,n));
    print_v_attr(v);
    printf("+++++++\n");
    //print_e_attr(g_list_nth_data(l,n));
}

//checking how file name errors is handled in readGraph function
gl=readGraph("");

```

```

//checking if it handles reading non xml files
gl=readGraph("graph_lib_test.c");

return 0;
}

```

../src/testonly/fileReadWriteTS.c

10 Garbage Collector

```

// author : Jing
#ifndef GC_H_NSBL_
#define GC_H_NSBL_

#include <glib/ghash.h>
#include <stdio.h>

typedef struct {
    void * ptr;          // object pointer
    int nref;            // number of references
    int type;            // type of object
} GC_Entry;

extern GHashTable * GCH;
void init_GC( GHashTable ** GChash );
void del_GC( GHashTable * GChash );
GC_Entry * GC_New_Entry( GHashTable * GChash, void * ptr, int type );
int GC_Ref( GHashTable * GChash, void * ptr, int type );
int GC_Deref( GHashTable * GChash, void * ptr, int type );
void GC_Out( GHashTable * GChash, FILE * out );

// IMPORTANT : use this wrapper
#define gcInit()          init_GC( &GCH )
#define gcDel()           del_GC( GCH )
#define gcRef(p,t)        GC_Ref( GCH, p, t )
#define gcDef(p,t)        GC_Deref( GCH, p, t )
#define gcOUT(o)          GC_Out( GCH, o )
#endif

```

../src/GC.h

```

// author : Jing
//
#include "GC.h"
#include <stdio.h>
#include <stdlib.h>

GHashTable * GCH;

void init_GC( GHashTable ** GChash ) {
    *GChash = g_hash_table_new( g_direct_hash, g_direct_equal );
}

void GC_Output_Entry ( gpointer key, gpointer value, gpointer stream ) {
    GC_Entry * gce = (GC_Entry *) value;
    FILE * out = (FILE *) stream;
    fprintf(out, "    | %12p | %4d | %4d |\n", gce->ptr, gce->type, gce->nref);
}

void GC_Out( GHashTable * GChash, FILE * out ) {
    fprintf(out, "GC |   pointer   | type | nref |\n");
    g_hash_table_foreach( GChash, & GC_Output_Entry, (void *) out );
}

void GC_Remove_Entry( gpointer key, gpointer value, gpointer dummy ) {
    GC_Entry * gce = (GC_Entry *) value;
    free( gce );
}

void del_GC( GHashTable * GChash ) {
    int ll = g_hash_table_size( GChash );
    if (ll!=0) {
        fprintf(stdout, "Warning: GC: member still exists.\n");
        GC_Out( GChash, stdout );
        g_hash_table_foreach( GChash, & GC_Remove_Entry, NULL );
    }
}

```

```

    }
    g_hash_table_destroy( GHash );
}

GC_Entry * GC_New_Entry( GHashTable * GHash, void * ptr, int type ){
    GC_Entry * gce = (GC_Entry *) malloc ( sizeof( GC_Entry ) );
    gce->ptr = ptr;
    gce->type = type;
    gce->nref = 0;
    g_hash_table_insert( GHash, (void *) ptr, (void *) gce );
    return gce;
}

int GC_Ref( GHashTable * GHash, void * ptr, int type ) {
    GC_Entry *gce = (GC_Entry *) g_hash_table_lookup( GHash, ptr );
    if (gce == NULL) gce = GC_New_Entry( GHash, ptr, type );
    int tt = ++gce->nref;
    // GC_Out( GHash, stdout );
    return tt;
}

int GC_Deref( GHashTable * GHash, void * ptr, int type ) {
    GC_Entry *gce = (GC_Entry *) g_hash_table_lookup( GHash, ptr );
    if (gce == NULL) {
        fprintf(stdout, "Error: GC: delete not existing member %p of type %d.\n", ptr, type);
        GC_Out( GHash, stdout );
        return 99;
    }
    if (gce->nref == 1) {
        g_hash_table_remove( GHash, ptr );
        free(gce);
    }
    return --gce->nref;
}

```

../src/GC.c

11 Makefile

```

BINDIR = ../bin
TESTDIR = ../devtest
MXMLDIR = ../mxmldir
LIBDIR = ../lib
INCLUDEDIR = ../include

CFLAGS = -D_NO_LOG
##### CFLAGS Options #####
# -D_AST_DEBUG
# -D_AST_DEBUG_ALL
# -D_NO_LOG
# -D_DEBUG
#####
XFLAGS = -7
##### XFLAGS Options #####
# -7 : 7bit ASCII
# -D_MYECHO : for LEX DEBUG
#####
YFLAGS =
##### CFLAGS Options #####
# -D_DEBUG
#####

GLIBLINK = `pkg-config --cflags --libs glib-2.0`
YACCLIB = -ly -lfl

CC = gcc -g $(CFLAGS) $(GLIBLINK)
LEX = flex
YACC = yacc
AR = ar

OBJ = Parser.tab.o LexAly.yy.o ASTree.o SymbolTable.o SymbolTableUtil.o util.o Error.o CodeGen.o CodeGenUtil.o

all : n2c.exe runlibs includes nsbl

n2c.exe : $(OBJ)
$(CC) -o n2c.exe $(OBJ) $(YACCLIB)
mkdir -p $(BINDIR)

```

```

mv n2c.exe $(BINDIR)
Parser.tab.o: Parser.tab.c Parser.tab.h
$(CC) -c Parser.tab.c
LexAly.yy.o: LexAly.yy.c Parser.tab.h
$(CC) -c LexAly.yy.c
Parser.tab.c : Parser.y
$(YACC) $(YFLAGS) -d Parser.y
mv y.tab.c Parser.tab.c
Parser.tab.h : Parser.tab.c
mv y.tab.h Parser.tab.h
LexAly.yy.c : LexAly.l
$(LEX) $(XFLAGS) LexAly.l
mv lex.yy.c LexAly.yy.c
ASTree.o : ASTree.c ASTree.h Parser.tab.h
$(CC) -c ASTree.c
SymbolTable.o: SymbolTable.h SymbolTable.c
$(CC) -c SymbolTable.c
SymbolTableUtil.o: SymbolTableUtil.c
$(CC) -c SymbolTableUtil.c
util.o: util.h util.c
$(CC) -c util.c
Error.o: Error.c
$(CC) -c Error.c
CodeGen.o: CodeGen.h CodeGen.c
$(CC) -c CodeGen.c
CodeGenUtil.o: CodeGenUtil.c CodeGenUtil.h
$(CC) -c CodeGenUtil.c
Derivedtype.o : Derivedtype.h Derivedtype.c
$(CC) -c Derivedtype.c
GC.o : GC.c
$(CC) -c GC.c
NSBLio.o: NSBLio.c
$(CC) -c NSBLio.c
FileReadWrite.o: FileReadWrite.h FileReadWrite.c libmxml.a
$(CC) -c FileReadWrite.c
libmxml.a :
cd $(MXMLDIR); ./configure; make libmxml.a; cd -
mkdir -p $(LIBDIR)
cp $(MXMLDIR)/libmxml.a $(LIBDIR)
runlibs: NSBLio.o Derivedtype.o FileReadWrite.o GC.o
rm -f libnsblgraph.a
$(AR) -cvq libnsblgraph.a NSBLio.o Derivedtype.o FileReadWrite.o GC.o
mkdir -p $(LIBDIR)
cp libnsblgraph.a $(LIBDIR)
nsbl : genScript.sh
sh ./genScript.sh
chmod +x ./nsbl
cp nsbl $(BINDIR)
includes : NSBLio.h Derivedtype.h FileReadWrite.h type.h operator.h
mkdir -p $(INCLUDEDIR)
cp nsbl.h NSBLio.h Derivedtype.h FileReadWrite.h GC.h type.h operator.h $(INCLUDEDIR)

testglib : Derivedtype.o graph_lib_test.c
$(CC) -c graph_lib_test.c
$(CC) -o $(BINDIR)/graph_lib_test.exe Derivedtype.o graph_lib_test.o $(GLIBLINK)

clean-mxml :
cd $(MXMLDIR) ; make distclean; cd -

clean :
rm -f Parser.tab.c Parser.tab.h LexAly.yy.c y.output *.o
rm -f Derivedtype.o graph_lib_test.o $(BINDIR)/graph_lib_test.exe
rm -f libnsblgraph.a

clean-include :
rm -f $(INCLUDEDIR)/*

distclean : clean clean-include clean-mxml
rm -f $(BINDIR)/n2c.exe $(BINDIR)/nsbl nsbl
rm -f $(LIBDIR)/libnsblgraph.a $(LIBDIR)/libmxml.a
rm -f -r $(BINDIR) $(LIBDIR) $(INCLUDEDIR)

```

../src/Makefile

```

#!/bin/sh
if [ $# -gt 0 ]
then
    PATH=$1
else
    cd ../
    PATH='pwd'

```

```

    cd src/
fi

echo "#!/bin/sh"
ROOT="\${PATH}"
CC="\gcc\"
CFLAG="-O2\"
BIN="\${ROOT}/bin\"
LIB="\${ROOT}/lib\"
INCLUDE="\${ROOT}/include\"
CLIB="-lpthread `pkg-config --cflags --libs glib-2.0`\"

if [ $# -eq 0 ]
then
    echo "ERROR: missing input file."
    exit 1
elif [ $# -gt 1 ]
then
    echo "ERROR: more than 1 input files."
    exit 2
fi

NSBLFILE="\${1}"
CFILE="\${NSBLFILE}.c\"
echo "\${BIN}/n2c.exe \${NSBLFILE}\"
\${BIN}/n2c.exe \${NSBLFILE}
if [ $#? -eq 0 ]
then
    echo "\${CC} \${CFLAG} \${CFILE} -I\${INCLUDE} \${LIB}/libnsblgraph.a \${LIB}/libmxml.a \${CLIB}\"
    \${CC} \${CFLAG} \${CFILE} -I\${INCLUDE} \${LIB}/libnsblgraph.a \${LIB}/libmxml.a \${CLIB}
fi
" > nsbl

```

../src/genScript.sh