COMP 6421 the Report of Assigment3

Semantic Analysis

Liao Xiaoyun

40102049 sharon.liaoxy@gmail.com

List of semantic rules implemented

Symbol table creation phase

1. [√] A new table is created at the beginning of the program for the global scope.
2. [√] A new entry is created in the global table for each class declared in the program. These entries should contain links to local tables for these classes.
3. [√] An entry in the appropriate table is created for each variable defined in the program, i.e. a class' data members or a function's local variables.
4. [√] An entry in the appropriate table is created for each function definition (free functions and member functions). These entries should be links to local tables for these functions.
5. [√] During symbol table creation, there are some semantic errors that are detected and reported, such as multiply declared identifiers in the same scope, as well warnings such as for shadowed inherited members.
6. [√] All declared member functions should have a corresponding function definition, and inversely. A member function that is declared but not defined constitutes an "no definition for declared member function" semantic error. If a member function is defined but not declared, it constitutes an "definition provided for undeclared member function" semantic error.
7. [√] The content of the symbol tables should be output into a file in order to demonstrate their correctness/completeness.
8. [√] Class and variable identifiers cannot be declared twice in the same scope. In such a case, a "multiply declared class", "multiply declared data member", or multiply declared local variable" semantic error message is issued.
9. [√] Function overloading (i.e. two functions with the same name but with different parameter lists) should be allowed and reported as a semantic warning. This applies to member functions and free functions.

Semantic checking phase – binding and type checking

10. [√] Type checking is applied on expressions (i.e. the type of sub expressions should be inferred). Type checking should also be done for assignment (the type of the left and right hand side of the assignment operator must be the same) and return statements (the type of the returned value must be the same as the return type of the function, as declared in its function header).
11. [√] Any identifier referred to must be defined in the scope where it is used (failure should result in the following error messages: “use of undeclared local variable”, “use of undeclared free function”, “use of undeclared class”).
12. [√] Function calls are made with the right number and type of parameters. Expressions passed as parameters in a function call must be of the same type as declared in the function declaration.
13. [√] Referring to an array variable should be made using the same number of dimensions as declared in the variable declaration. Expressions used as an index must be of integer type. When passing an array as a parameter, the passed array must be of compatible dimensionality compared to the parameter declaration.
14. [√] Circular class dependencies (through data members\inheritance) should be reported as semantic errors.
15. [√] The“.”operator should be used only on variables of a class type. If so, its rightoper and must be a member of that class. If not, a “undeclared data member” or “undeclared member function” semantic error should be issued.

All the above semantic rules were implemented in my project.

Design

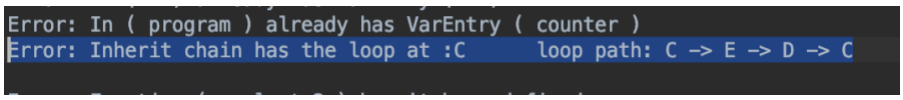
1. I used the astvisitor provide by the professor posting on the course website. This tool has a well useful and flexible structure for gathering, aggregating and verifying syntactic information in the AST.
2. Converting the AST generated in assignment 2 to a new AST constructed by the node defined in the astvisitor, the type of nodes in the as visitor isn't enough, then I added many new types of nodes to support my AST.
3. Implementing SymTabCreationVisitor, almost each node has a visit function in SymTabCreationVisitor.

    1) Creating entries and tables
    Creating an entry for each class, function, declared variable, parameter, inheritance. Creating a local table for each class and function including main function.

    2) Multiply declared function and class
    By looking up in entries and local tables to prevent multiple declaring. If two functions with the same name and same parameter list, then will report multiply declared error.

    3) Function overloading
    If two functions with the same name but with different parameter list, then will report a warning of function overloading. I overwrite the equal function in the VarEntry, if two parameters are same, then their types and dimensions must be the same.

    4) Linking inheritance classes and checking circular class dependencies
    Linking the inheritance class by looking up class in the global table and recording all inherit relations among the classes in a hashmap<classA, classB>. In case some inheritance class declared later than the inherit declare, I can check the hashmap when each class is declared and make up the missing inherit link. When the traversal of the AST is done, checking which inherit link is empty, and checking if there are circular class dependencies by deep searching the graph constructed by hashmap.



5) Traverse Ending checking
When the traversal of the AST is done, I will do 3 types of checking.

```
public void endingChecking(){
    checkAllInherit();
    checkInheritCircle();
    checkAllFuncDeclare();
}
```

- checkAllInherit(), checking if all the inheritance data member have the link to the inheritance class.
- checkingInheritCircle(), check if there is a circle of inherited classed.
- checkAllFuncDeclare(), check if all declared functions have be defined. We need to check which function haven't be defined. And for preventing multiply define function, I set a define flag to track the define action, if flag is true, means this function has been defined.

```
public void checkAllFuncDeclare(){
    System.out.println("total member function:" + funcDeclareList.size());
    for (FuncEntry entry : funcDeclareList){
        if (entry.definedFlag == false){
            String paramStr = covertParamlsitToStr_varEntry(entry.getParamsEntryList());
            System.out.println("Error: Function ( " + entry.m_name+ " ) hasn't been defined.");
            System.out.println("       func "+ entry.m_name+ " ( "+paramStr + " ) : "+ entry.m_type +"\n");

            this.m_errors += "Error: Function ( " + entry.m_name+ " ) hasn't been defined.\n"
                    +"       func "+ entry.m_name+ " ( "+paramStr + " ) : "+ entry.m_type +"\n";

        }
    }
```

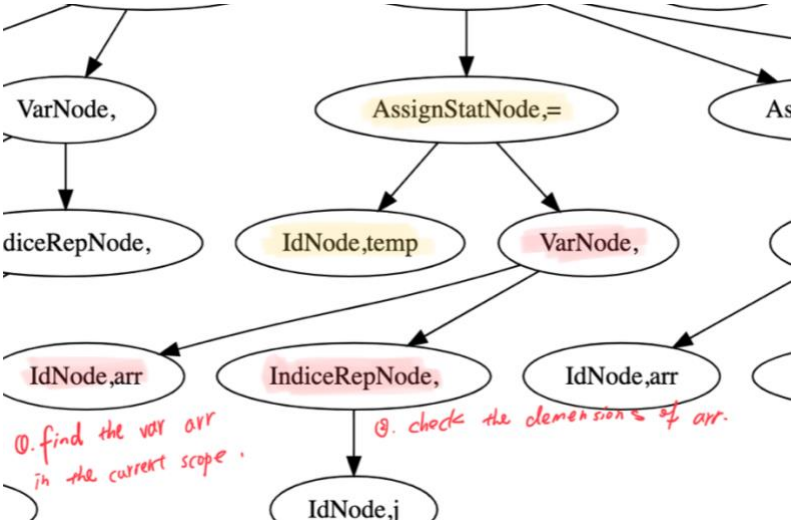
4. Implementing TypeCheckingVisitor, almost each node has a visit function in TypeCheckingVisitor to check its type.

1) Type checking on the expression
For checking if the type of the elements of the expression is valid, I need to know the type of element by traversing the subtree of the elements. So, to implement the type checking of expression, I will implement the type checking of variable and function call first, there are the basis of expression.

```
public void visit(AddOpNode p_node){
    // idNode, NumNode, varNode, funcCall, add, mlt
    for (Node child : p_node.getChildren() )
        child.accept( p_visitor: this);

    String leftOperandType  = p_node.getChildren().get(0).getType();
    String rightOperandType = p_node.getChildren().get(1).getType();

    if( leftOperandType.equals(rightOperandType) ){
        p_node.setType(leftOperandType);
        VarEntry entry = new VarEntry( p_kind: "var",leftOperandType, p_name: "",new Vector<Integer>());
        p_node.m_symtabentry = entry;

    } else{
        p_node.setType("typeerror");
```

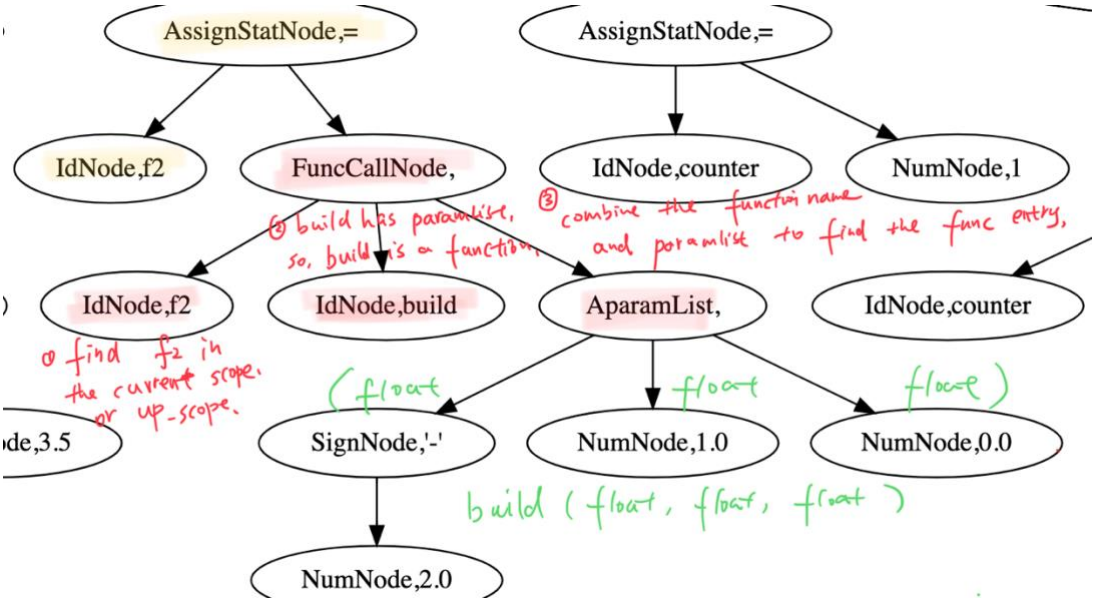
```
public void visit(AssignStatNode p_node){
    for (Node child : p_node.getChildren() ){
//          System.out.println("child : "+ child.getClass().getSimpleName());
        child.accept( p_visitor: this);
    }

    String leftOperandType  = p_node.getChildren().get(0).getType();
//  System.out.println("leftOperandType : "+ leftOperandType);

    String rightOperandType = p_node.getChildren().get(1).getType();
    if( leftOperandType.equals(rightOperandType) ){
        p_node.setType(leftOperandType);
        VarEntry entry = new VarEntry( p_kind: "var",leftOperandType, p_name: "",new Vector<Integer>());
        p_node.m_symtabentry = entry;
    } else{
        p_node.setType("typeerror");
```

2) Type checking on the variable
   Variable node has two types of children, Id node and indices node.
   To get the type of variables, we should check the type of id node and indices node.



   For the above example, the var node has two children, one is id node, the other is indice node. First, find the entry of var arr in the current scope, then check its dimension is corresponding to the indice node. If two nodes' information are correct, then we can confirm the type of var node is the type of id node.

3) Type checking on the function call



Function call node has three types of children, IdNode IndiceRepNode AparamList. To check the type of function call node, need to consider the order of the children.

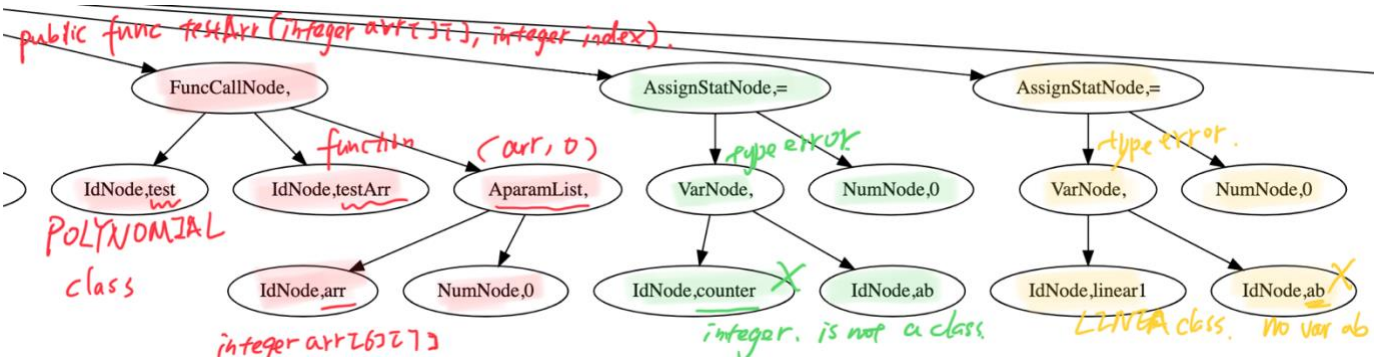
- If the first Id node is a function, this function will be a free function.
- If the previous node of id node is also an id node, then this id should be a class name.
- If the follow node of id node is an aparamlist node, then this id should be a function name, because we allow the function overloading, so we need combine the function name and param list to find the corresponding function.

```
public FuncEntry lookUpFunc(String funcName, String returnType, ArrayList<VarEntry> paramList){
    FuncEntry funcEntry = null;

    for(SymTabEntry entry : this.m_subtable.m_symlist){
        if(entry.getClass().getSimpleName().equals("FuncEntry")){

            ArrayList<VarEntry> funcDeclareParams = new ArrayList<>();
            for(SymTabEntry subentry : entry.m_subtable.m_symlist){
                if(subentry.m_kind.compareTo("param") == 0){
                    funcDeclareParams.add((VarEntry) subentry);
                }
            }

            boolean nameEq = funcName.equals(entry.m_name);
            boolean paramEq = funcDeclareParams.equals(paramList);


            if(nameEq && paramEq){
                return (FuncEntry) entry;
            }
```

- The type of function call be void.

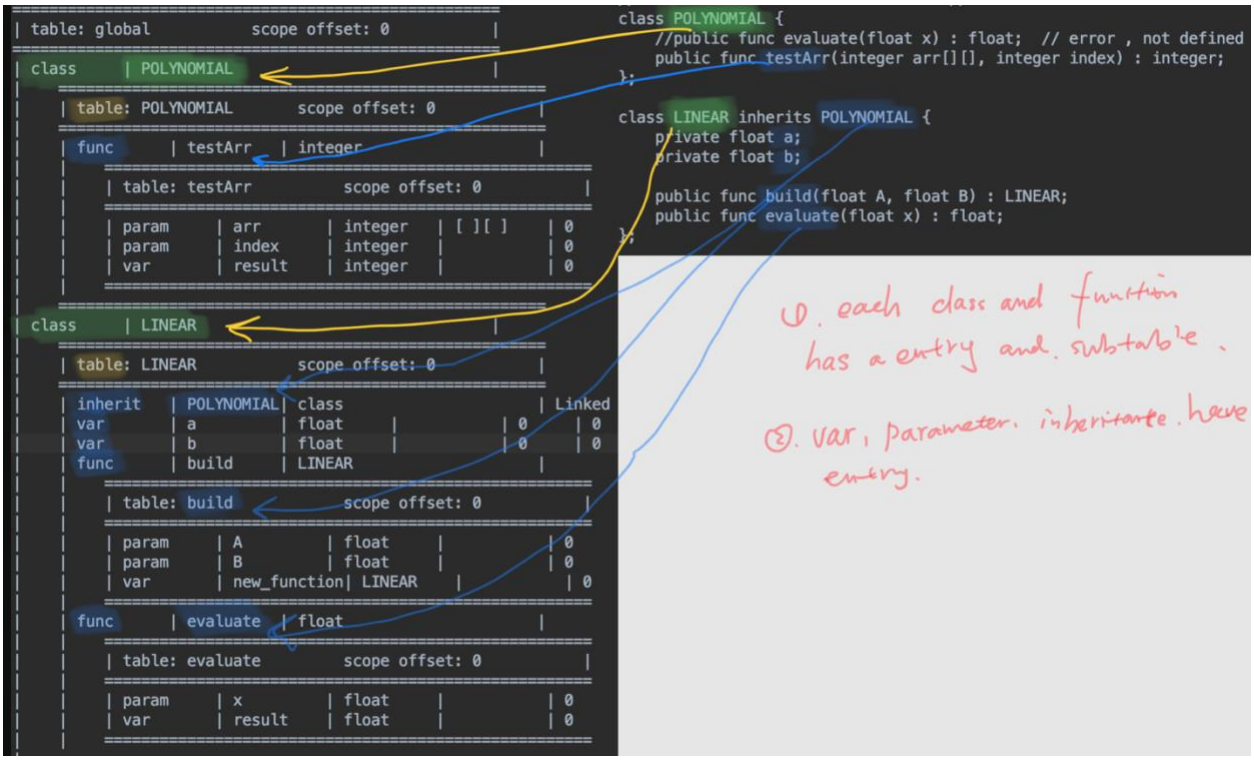
4) Type checking example

```
class POLYNOMIAL {
   public func testArr(integer arr[][], integer index) : integer;
};

main
{
  var
  {
    POLYNOMIAL test;
    integer arr[6][7]; // can modify dim to test function call
  }
  test.testArr(arr,0); // testing function call

  counter.ab = 0; // error, count isn't a class
  linear1.ab = 0; // error, class LINEAR doesn't have var ab
}
```



Test case

1. Symbol Table



2. Overloading and circular class dependencies

```
class QUADRATIC inherits POLYNOMIAL {
    private float a;
    private float b;
    private float c;

    public func build(float A, float B, float C) : QUADRATIC;
    public func evaluate(float x) : float;
    public func evaluate(integer x) : float; // warning, overloading
};

class CircularTest inherits AA {  // error, Circular class dependencies

};

class AA inherits BB {

};

class BB inherits CircularTest {

};
```

```
Warning: Overloading:  function ( evaluate ) overloading function in (  QUADRATIC ) class
Error: Inherit chain has the loop at :CircularTest      loop path: CircularTest -> AA -> BB -> CircularTest
```

3. Type checking on expression, variable and function call

```
main
{
  var
  {
    LINEAR linear1;
    QUADRATIC quadratic2;
    integer counter;
    float floatCounter;
    POLYNOMIAL test;
    integer arr[6]; // can modify dim to test function call
  }
  linear1 = linear1.build(2, 3.5);  // error,need build(float,float)
  quadratic2 = quadratic2.build(-2.0, 1.0, 0.0);

  counter = 1; // integer counter
  floatCounter = 1.0;
  test.testArr(arr,0); // testing function call

  counter.ab = 0; // error, count isn't a class
  linear1.ab = 0; // error, class LINEAR doesn't have var ab

  while(counter <= 10)
  {
    write(counter);
    evaluate(counter); // error,  param must be float
    linear1.evaluate(counter); // error, evaluate(float x)
    linear1.evaluate(floatCounter); // good
    quadratic2.evaluate(counter); // good, call overloading function
  };
}
```

int

arr should be arr[][]

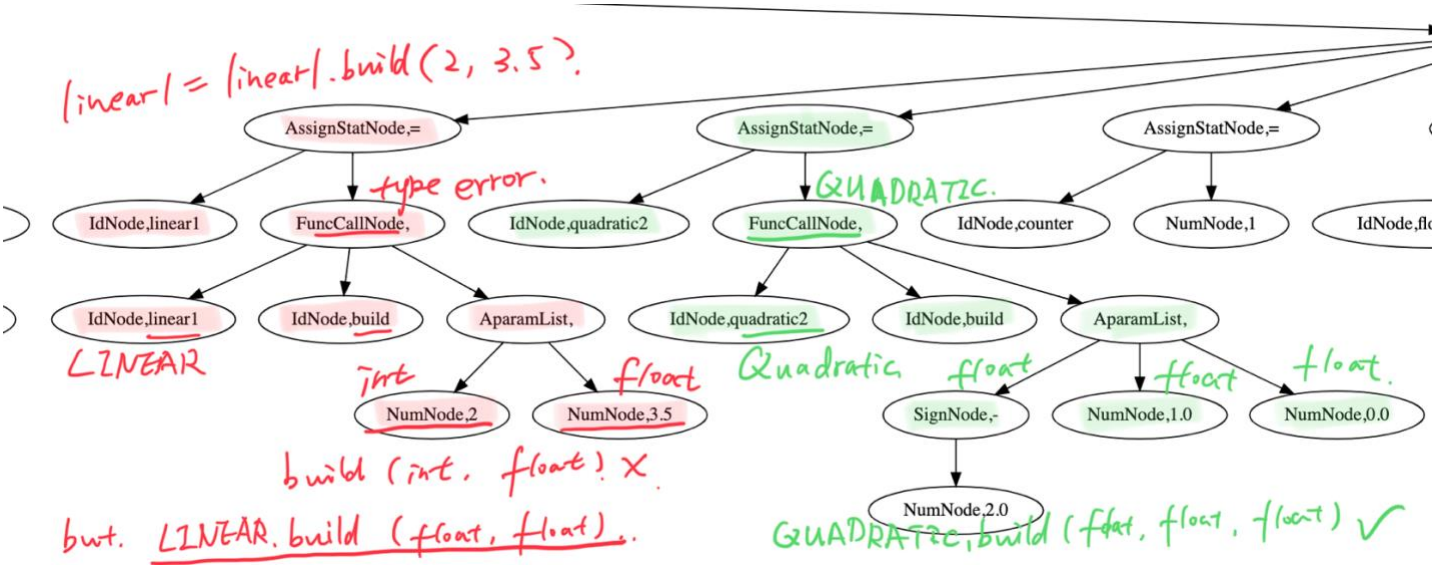
```
Fucntion call error:  linear1.build(2,3.5),  build( integer,float, )
AssignStatNode type error: linear1=linear1.build(2,3.5);,  linear1(LINEAR) and (TypeError)
Fucntion call error:  test.testArr(arr,0),  testArr( integer,integer, )
Var error:  counter.ab,  Can't find the  class ( counter ) , use of undeclared class
AssignStatNode type error: counter.ab=0;,  (TypeError) and 0(integer)
Var error:  linear1.ab,  Can't find the var : (ab )  in the class ( LINEAR ), use of undeclared
AssignStatNode type error: linear1.ab=0;,  (TypeError) and 0(integer)
Fucntion call error:  evaluate(counter),  evaluate( integer ), can't find this free function
Fucntion call error:  linear1.evaluate(counter),  evaluate( integer, )
```

type error.

```
|   ===================================================
| func        | evaluate  | float                      |
|   ===================================================
|    | table: evaluate        scope offset: 0          |
|   ===================================================
|    | param     | x         | float    |          | 0
|   ===================================================
| func        | program   | void                       |
|   ===================================================
|    | table: program         scope offset: 0          |
|   ===================================================
|    | var       | linear1   | LINEAR    |         | 0
|    | var       | quadratic2| QUADRATIC |         | 0
|    | var       | counter   | integer   |         | 0
|    | var       | floatCounter| float    |         | 0
|    | var       | test      | POLYNOMIAL|         | 0
|    | var       | arr       | integer   | [6]     | 0
|   ===================================================
===================================================
```

free func

main.



Tool

1. https://dreampuf.github.io/GraphvizOnline

   This website can help me to generate a readable AST.

2. Astvisitor

I used this patten to construct my symbol table and type checking.