1. **List of semantic rules implemented**

Checklist:

1. |X| global symbol table

2. |X| class symbol table

3.1 |X| class data member

3.2 |X| function's local variable

4.1 |X| free function

4.2 |X| member function

6.1 |X| undeclared member function definition

6.2 |X| undefined member function declaration

8.1 |X| multiply declared class

8.2 |X| multiply defined free function

8.3 |X| multiply declared identifier in class

8.4 |X| multiply declared identifier in function

8.5 |X| shadowed inherited data member

9.1 |X| Overloaded free function

9.2 |X| Overloaded member function

9.3 |X| Overridden inherited member function

10.1 |X| Type error in expression

10.2 |X| Type error in assignment statement

10.3 |X| Type error in return statement

11.1 |X| Undeclared local variable

11.2 |X| Undeclared data member

11.3 |X| Undeclared member function

11.4 |X| Undeclared free function

11.5 |X| Undeclared class

12.1 |X| Function call with wrong number of parameters

12.2 |X| Function call with wrong type of parameters

13.1 |X| Use of array with wrong number of dimensions

13.2 |X| Array index is not an integer

13.3 |X| Array parameter using wrong number of dimensions

14.1 |X| Circular class dependency

15.1 |X| "." operator used on non-class type

1. **Design**
2. Overall

Class diagram as follows:

Diagram

Description automatically generated

I used the visitor pattern to do abstract syntax tree traversal for several phases: constructing subtree representation, generating symbol tables and checking types.

Three concrete visitor classes as well as the base visitor class are created. The **ReconstructSourceProgramVisitor** class is to construct a string the represents the subtrees. The **SymTabCreationVisitor** class is to create symbol tables and symbol entries. The **TypeCheckingVisitor** class is to compute the type of subtrees and check type bindings.

The **SymTab** class and **SymTabEntry** class represent symbol table and symbol entry. Concrete SymTabEntry classes, including **ClassEntry, FuncEntry, VarEntry** are created to represent symbol entries for classes, functions and variables.

The class of **SematicAnalyzer** is responsible for dealing with every file. Having three visitor objects to do tree traversal starting from the root node(progNode) in three phases.

The **SemanticAnalyzerDriver** opens every file(.src) in a folder or a single file(.src), first creates a SyntacticAnalyzer object and then a SemanticAnalyzer object to handle the task of every visitor and output reports to files.

More detailed explanation can be seen in the source code.

1. **Constructing subtree expression**

This sematic action is useful for debugging and informative error reporting, exposing the location and expression of the errors. It mainly aims represent to nodes that has expression meaning, but many nodes need to be traversed to transmit the information. In fact, the progNode, for example, has a m\_subtreeString which contains all source tokens.

It maps to all AST nodes.

1. **Symbol table creation**

This phase is to create symbol tables and their entries. Almost all AST nodes are used to transmit symbol table information.

Some nodes represent identifier declarations/definitions in the form of a symbol table record to be inserted in a symbol table.

VarDeclNode -> variable entry

ClassDeclNode -> class entry

FuncDeclNode -> function entry

FuncDefNode -> free function entry

MainBlockNode -> main function entry

Some nodes represent a scope, in which case they need to create a new symbol table, and then insert the symbol table entries they get from their children.

ProgNode -> global symbol table

ClassDeclNode -> class table

FuncDefNode -> function table

MainBlockNode -> main function table

1. **Binding and type checking**

This phase computes the type of subtree expressions, assignment statements, return statements, etc. and do some semantic checking like array dimension consistence, function calls parameters, etc.

ClassDeclNode -> Circular class dependency

DimListNode -> Array dimension

ReturnStatNode -> Type error in return statement

AssignStatNode -> Type error in assignment statement

InlineIfStatNode -> Type error in inline if statement

AddOpNode -> Type error in addition operation

MultOpNode -> Type error in multiplication operation

ExprNode ->Type error in expression

VariableNode -> Use of array with wrong number of dimensions

TypeNode -> Use of undeclared class

IdNode -> Undeclared local variable

DotNode -> Use of array with wrong number of dimensions/ Undeclared member function/ Undeclared data member

FuncCallNode -> Undeclared free function/ Function call with wrong type(number) of parameters/ Array parameter using wrong number of dimensions

IndiceNode -> Array index is not an integer

1. **Use of tools**

I used the **visitor pattern** following the codes example the professor provided, which gives me a general idea to implement the pattern in this assignment. Since different semantic actions are needed to do semantic checking and the next assignment also contains semantic actions, this pattern separated different group of operations, making the implementation much easier.

As for writing codes, I used [IntelliJ IDEA](https://www.jetbrains.com/idea/), which is great at coding auto completion, code debugging, etc. The class diagram is generated by [PlantUML](https://plantuml.com/), which can be integrated in the IDE conveniently using a plugin called [sketch it](https://plugins.jetbrains.com/plugin/10387-sketch-it-). Through this workflow, the drawing of UML diagram is greatly simplified.