Semantic Analysis Report

Irfan Ali - CS16BTECH11019
Sudhanshu Chawhan - CS16BTECH11037

Introduction :

We had to create a semantic analyzer for COOL(Classroom Object Oriented Language). The semantic analyzer has to reject programs which are erroneous semantically and type annotate the AST for correct programs.

Basic Design :

Our semantic analyzer gets the AST built by the parser. We traverse the AST and do the below mentioned things.

1. Build the Inheritance Graph
2. Check if the Inheritance Graph has cycles
3. Traverse Inheritance Graph and check semantics.

Inheritance Graph :

The inheritance graph is stored as a hashmap which stores String className vs AST.class_ class. In the first pass of the AST, all the classes are stored in the inheritance graph. The standard classes - Object, IO, Int, Bool, String are also stored in the inheritance graph. Then checkGraph() function is called which checks if there exists any cycle.

To assist in checking for cycles and other semantic checking, a hashmap classTable is created which stores name of a class vs the name of its parent class. All the classes are inserted into the classTable. If the name of the class to be inserted is same as a Standard Class, an error is reported and the class is not inserted in the table. We also check for redefinition of classes before inserting in the classTable. Skipping the insertion in the classTable is the recovery method used here.

For checking cycles, we iterate over the classTable and for every class, we check all its ancestor classes. If any ancestor class is same as the given class, there exists a cycle. Error message is reported and the program terminates. Other semantics like cannot inherit from Int, Bool, and String are also checked here.

In AST.java, a list of strings is added to AST.class_ which stores the list of names of the child classes. After the checkGraph() method , another method setChildren() is called in the inheritance graph constructor which sets the children of each class. Here again the classTable is iterated over and for all classes in the classTable, we go to its parent in the inheritance graph and the class as its child.

Now the inheritance graph is ready.

Name Mangling :

COOL allows methods which are defined in a class to be redefined in an inherited class but only the method body can be changed. The return type and parameters cannot be changed. To implement this feature, we need name mangling, which means adding information about the parameters in the function name.

The mangled name of a function is created in the given manner:

1. Class name followed by a dollar symbol is added to the mangled name.
2. Then function name followed by dollar symbol is added.
3. Then "_NP" followed by number of parameters followed by dollar symbol is added.
4. Then all the parameter types separated by dollar symbols are added.

Ex: For a function like "foo(a : Int, b : String)" in class Cl , the mangled name will be " Clfoo_NP2IntString$ ".

The return type is not included in the mangled name. A hashmap named methodReturnTable stores mangled name of a function vs its return type. This is used in checking if a function is redefined with different return type.

After the checking of inheritance graph is done, a method is called which stores the mangled names of all the functions and their return types in methodReturnTable.

Traversing Inheritance Graph :

After the mangled names are stored, the inheritance graph is traversed to check the semantics. A Depth First Search is applied on the inheritance graph starting from the root class (Object).

The traverse function of the inheritance graph applies DFS on it by starting from the root node. It calls the visit function for the class and all its children.

The visit function of a class visits all its features. All the visit functions do the type checking and semantic checking. If any error is encountered in type checking, the type is set to "Object" and the analysis continues. This is the recovery method.

To maintain the scope, two scopeTables are used. One maintains the scope for attributes and other is for methods. The visit functions also check the scope.

File Structure :

The semantic analyser is divided into different classes to make the code more organized and readable.

1. *AST.java* : Contains all the node classes of the AST. Very few changes have been made to this file.
2. *InheritanceGraph.java* : This contains the InheritanceGraph class and all the functions related to the inheritance graph.
3. *Visitor.java* : This contains the Visitor class. It contains all the visit methods which help in checking the semantics by visiting the AST nodes.
4. *GlobalData.java* : This contains the GlobalData class. It contains all the variables, constants and objects which have been used throughout the semantic analyzer. Ex : It stores the inheritance graph object , the scope tables etc. It also has methods which have been frequently used like methods to get mangled name of a function.

Rest all files are left unchanged.

Test Cases :

Test cases have been added in the testcases folder. Both good and bad test cases have been added. The test cases have been commented to show which rule has been violated.