Semantic Analysis: CS3423

Programming Assignment - Semantic Analysis of Cool Language

Report and Documentation

Sagar Jain CS17BTECH11034
Sai Harsha Kottapalli CS17BTECH11036

Table of Contents

Stages of Analysis

Overview

A given cool file goes through the following stages in the assignment:

1) Lexer (given)

2) Parser (given, outputs the AST)

3) Semantic Analysis (Implemented)

Semantic analysis is done by us in the following steps:

1) Utilising the AST to get information about classes present, their parents, etc to construct an **Inheritance Graph**. This graph contains the class **Object** as its root node and all the other classes with edged from parents to their children classes. Analysis related to inheritance, class definitions, etc is done here.

2) Once again, we use the class information within the AST to summarise all the information about the classes and methods in them to create a **Class Table**.

3) Finally, we traverse the AST recursively while creating a **Symbol Table**, while creating the symbol table all kinds semantic properties as mentioned in the cool manual at https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf are checked and if any property does not hold an error is reported along with an effort to recover.

There is an ErrorReporter class as well which allows us to print out errors in a uniform pattern.

Code Explanation

Creating The Inheritance Graph

The first high-level task mentioned in the assignment is to *Look at all classes and build an inheritance graph.*

This part of the assignment was done in the file **InheritanceGraph.java**.

This task was accomplished by us through the class **InheritanceGraph.** The following are the steps in which the inheritance graph is built by the constructor of the class which takes a single argument of **AST.program**:

1) Addition of the basic classes (as the last two nodes) to the graph i.e. Object & IO. Bool, Int, String are not added since they cannot be inherited from or redefined.
2) Adding edges to the graph i.e. from parent classes to children classes. This is done by looping through **program.classes**. Since these are the only edges possible, graph construction is now complete.

Points to Note about the Inheritance Graph:

1) The graph is represented in the form of an adjacency list.
2) The nodes are essentially just index numbers of classes, translation of index numbers to class names is done by using hash maps in both the directions, namely **classIDs** and **iDName**. This choice was made since string comparision while constructing the graph could be avoided.

While constructing the graph tests are performed before the insertion of every node and every edge. For example, **redefinition**, **undefined parent**, **inheriting from** restricted classes, etc.

The second high-level task mentioned in the assignment is to *Check that the graph is well-formed.*

A part of this task is accomplished by the tests done while building the graph, the other major portion of this task is to check if the graph contains any cycles since cool does not allow cyclic inheritance. This check is done by using DFS to search for back edges since the existence of a back edge implies the existence of a cycle. The functions used for this task are **containsCycle** and **checkBackEdge**.

Creating The Class Table

This part of the assignment was done in the files:

1) ClassNode.java
2) ClassTable.java

The next part of the assignment involves creating a table of ClassNodes which are similar to the AST class_ nodes but contain much more information (like parent attributes). Creation of this table not only helps us check various semantic properties, like parent-child function redefinition, etc. but at the same time, the creation of the symbol table also uses the class table. The table is built in the following steps:

1) We first create the nodes for the basic classes, by filling their information and then insert them into the table.
2) Then we traverse over the inheritance graph in a breadth-first way and enter every class one by one along with all their attributes & methods.

Points to Note about the Class Table:

1) We need to traverse the inheritance graph in a breadth-first way because we do not want to insert any child class before the parent, if this happens there we might miss out on catching a few errors. We would also have incomplete information about the child classes as they would not be able to inherit any features from their parent classes.

2) While inserting any class into the class table, several semantic properties of cool are tested, to name a few:
    a) Redefinition of methods in the same class.
    b) Invalid redefinition of inherited methods.
    c) Invalid return types, arguments, etc.

Primary data structures used are **ClassNode, cnHm**(class node hashmap), **classScope**.

Creating The Symbol Table

This part of the assignment was done in the files:

1) ScopeTable.java
2) ScopeTableImpl.java

This part of the assignment involves few of the most important semantic tasks i.e. **type checking**, **building the symbol table** & **annotating the AST**.

We have used the default implementation of the scope table given to us for this assignment. The scope table is a template class, the template argument we use is **AST.attr**. So our entires into the symbol table are of the type $<$name, AST.attr$>$.

This task is accomplished in the following steps:

1) The constructor of the SymbolTableImpl class takes an AST.scope argument. It starts off by filling the class table (explained above) following which it starts building the symbol table.
2) We first insert the **self** variable for every class into the symbol table.
3) For every class, all its attributes are inserted into the scope table. Following which they are individually traversed by entering the class scope.
4) For every class, all the attributes are entered into the scope table recursively. Within a class, we use the visitor pattern to keep traversing till the leaf nodes of the AST and inserting them into the scope table while also checking the semantic properties of cool.
5) Every node has different types of semantic checks to be performed. We use a uniform pattern to do this which is:
    a) A call to **traverseNode** method is made with some AST.expression as argument.
    b) Check for the type of this argument using chained if else statements.
    c) If class is found cast this expression node into the expected type.
    d) Run **traverseNode** on all the subexpressions of this expression.
    e) Using the new found information from above run all semantic checks for this type of node.
    f) Annotate the node with the type of the expression.

Points to Note

1) Error reporting with regards to the cyclic inheritance can be made more robust by including instances of cycles found in the inheritance graph.
2) A subset of dispatch calls is not handled in our implementation because of avoiding the SELF_TYPE type.
3) There was a bug in the parser with respect to the creation of the AST.neg and AST.comp node (the two were flipped). // our implementation assumes the correct AST.
4) Scoping for local, global values are handled by the variable **scope** in the ScopeTable data structure which keeps track of the current scope depth and all the parent scopes are stored in the ScopeTable ArrayList.

Explanation of Test Cases

Bad Testcases

1. Use of Base Classes is not allowed - (Int, Bool, Object, etc).
2. Error in dispatch node, no method with given name in the class.
3. Inheriting undefined class
   Class 'Main' is missing
4. Method 'main' is missing in 'Main' class.
5. Multiple Class definitions not allowed
6. Inheriting from Bool, Int, String is not allowed
7. Attribute redeclaration not allowed
   Method redeclaration not allowed
   Use of undeclared variables
8. Incorrect arguments size of child class w.r.t. Parent Class.
   Incorrect return type of child class w.r.t. Parent Class.
9. Incorrect argument type of child class w.r.t. Parent Class.
   Incorrect return type of method
10. Method parameter redeclaration is not allowed.
11. Complement cannot be applied on non-Int type
    Negation cannot be applied on non-Bool type
    LHS and RHS of '<', '<=' operator have to be equal to Int
    'new' can be applied to declared classes only.
    Expression type should be same as identifier's type to which assignment is being
    done.
12. 'let' typeid must be equal to its value's type
    Predicates in loop/if must evaluate to a Bool
13. When assignment is being done, LHS and RHS must have same types
14. Check that add, sub, div, mult should have Int type operands only.
15. Error in typcase node, two branches with same type.