Semantic Analysis Assignment

Atharva Sarage - CS17BTECH11005
Yash Khasbage - CS17BTECH11044

1. **Introduction**:

In this assignment, we code a semantic analyzer for the Cool language. Cool is an object-oriented compiler-based language. The features of loops, variable declaration, conditional statements, string manipulation, etc. make it a Turing complete language. Another feature of the Cool language is SSA or Single Static Assignment. SSA allows one variable to be assigned only once in its lifetime. In other words, you cannot reassign the value of a variable. Once set, the value of the variable is fixed. Being an Object-Oriented language, it allows one to declare classes and thus create objects. This feature also results in what is popularly known as "data hiding". The data/method belonging to one class is not accessible by an object of any other class(except for objects of inherited classes). With a fine analysis, a programmer can split the problem statements into a beautiful hierarchy of classes. This again forms the backbone of abstraction and code maintenance.

The word "semantic" refers to the information in the program/language that is more related to the meaning conveyed by the program. It can also be said that the semantic and syntactic features of a language form the language itself. The above mentioned were some semantic properties of the COOL programming language. It forms the task of a back-end of the compiler to actually check if a program follows the semantic rules specified by the language specification.

Our semantic analyzer is mainly concerned about these two topics:

1. type checking
2. inheritance graph

There are certain other semantic checks also which do not fall under a broad category. Details about these features will be provided in the next sections. The "SELF" type is not handled as mentioned in the assignment.

2. **Input and output for our semantic analyzer**:

A typical semantic analyzer needs an AST(Abstract Syntax Tree) as input. An AST consists of nodes, which represent different constructs of the source code. A simple example of such a node is the "int_const" node. This node will store an integer constant. It is always beneficial to store the program file name and its location line number. Storing the location of a construct helps in reporting errors.

The semantic analyzer traverses the AST in a node-by-node fashion. This traversal forms a way to collect information regarding all objects, class attributes and class methods. This is the information analyzed by a semantic analyzer. The semantic analyzer checks for various forms of semantic correctness. An example of a typical semantic check is the “Integer class cannot be inherited”. Other examples include checking type compatibility in LHS and RHS of equality.

Such semantic checks may or may not be violated by a program, depending on language specification. If the program fails some checks, the semantic analyzer provides an “Error message” regarding the check that failed. The line number and file name of the code that actually caused the error is also printed. At times, a smartly thought error message an be of use in detecting the semantic violation in the program and help in correcting it.
Thus, if a program violates some semantic checks, the semantic analyzer will give out error messages. But if the program passes all semantic checks, the semantic analyzer will simply return a flag that indicates the correctness of the program without any warning or error printed.

It must be noted that sometimes, all errors cannot be detected at the same time. For example, while detecting cycles in the inheritance graph, the semantic analyzer has to stop execution if a cycle is detected. This has to be done so that the further steps related to class inheritance has do not create an infinite loop when stuck in those cycles.

3. Structure of InheritanceGraph.java

There are 2 classes one for vertex and another for Graph itself.

Vertex class
Each member of the vertex class has a name, List of all its children, and its parent node
Utility methods such as getName, addNeighbours are implemented

Graph Class

A member of graph class has a parent-map which gives the parent of the current node.
List of all vertices in the graph and a nameToVertex map to get the vertex from class name.
Some utility functions such as addEdge, getVertex, check whether a class is present are implemented.

Cycle Detection

We use colour labeling to detect a cycle in our graph. Colour of a vertex is 0 if it is unvisited, it is 1 if the vertex is in the recursion stack and it is 2 if the vertex is already visited. We do a normal DFS traversal of our tree . Say we are at some vertex in our tree , there are some vertices in the recursion stack and now we iterate over the adjacency list of a current vertex. If we find that a vertex has color 1 which implies it is still currently presently in recursion stack and not popped(it it were popped its colour would have been 2) so we have a cycle. We return all the elements present in recursion stack to throw an error.

Least Common Ancestor

This method returns the least common ancestor of 2 nodes in the inheritance graph. Given the 2 nodes we take the parent pointers unless we reach NULL and simultaneously push the nodes in the stack. We do this for both the nodes, as a result, we have 2 stacks which store the vertices from root to node(root node is at the top of stack since LIFO property of a stack). Now we pop from both the stacks simultaneously and also store the popped element. When for the first time the top elements of the 2 stacks differ we know that the previous element was the LCA of the two nodes in question so we return that node. (Last node lying on the common path from root to both the nodes)

Conforms

This method takes 2 two nodes and tells if they are conforming. We take the parent pointers using parentNameMap until we get to the other node in this case we return true otherwise if we do not encounter the other node and we reach NULL we return false. None of the nodes ancestors matched the given node so they do not conform.

4. **Structure of ClassInfo.java**

ClassInfo has an InheritanceGraph object which will be populated as we would traverse the AST, a map from class name to AST class_ and information regarding scopes of attributes and functions.
It contains methods to initialize all the basic classes such as Int, Bool, IO, Object, String and a method to update the environment for a given class.

5. **Structure of Semantic.java**

Semantic class has a classInfo object and an error flag to report whether semantic check was successful or not.
There are 4 steps for semantic analysis -

1. collectAndValidateClasses(program)
   Above function call collects all class names. traverse through all classes. collect all attributes and all other information. Inheritance graph is also populated in this step.

2. runCycleReporter(program)
   Above function call checks for cycles in inheritance graph. If cycle is found it prints all The nodes of inheritance graph involved in the cycle and then exits with return value 1.
3. checkForUndefinedParent(program)
   Above function call checks if the the present is defined and that a class is not inheriting from an undefined parent

4. analyzeClassFeatures(program)
   Above function call analyses whether all class features are valid or not. Errors such as class redefinition, redefinition of attributes, and checks such as the existence of Main class, the main method in main class, etc.

5. After doing all these, a call is made to recurseTypeChecker(program) for further type checking.

6. **Structure of TypeChecker.java**

It contains various overloaded constructors one for each different possible node.
There are basically 2 classes of nodes expression nodes and non-expression nodes.
One constructor recursively calls other constructors creating a new object and appropriately passing the arguments.

7. **Structure of CoolUtils.java**

This class contains few string utility variables which are used in other files(they are declared static no need to create coolUtils object) and few methods such as createNewObjectScope, attrType which returns the type of attribute.

TEST CASES-

1. badTest1.cl - Cycle in Inheritance Graph
2. badTest2.cl- Inheritance from an undefined class.
3. badTest3.cl - Inheritance from Int,Bool class
4. badTest4.cl- Object created of an undefined class
5. badTest5.cl - Main() Class absent
6. badTest6.cl - main() method in Main() class absent
7. badTest7.cl - redefining attribute from inherited class
8. badTest8.cl - using undefined attribute
9. badTest9.cl - method body does not conform to return type
10. badTest10.cl - Multiple definition of classes
11. badTest11.cl - Predicate of condition of loop not Bool
12. badTest12.cl - Parameters in main() method
13. badTest13.cl - type of initialization in let does not conform to declared type
14. badTest14.cl - Redefining a method in derived class with different signatures but same name.
15. badTest15.cl - Incorrect types in arithmetic expressions
16. badTest16.cl - Class Inheriting itself

17. goodTest1.cl - Inheritance graph has no cycles
18. goodTest2.cl- Types of arithmetic expressions
19. goodTest3.cl- Types of identifiers
20. goodTest4.cl- Types of constants
21. goodTest5.cl- Conformance checking