

CS126: Introduction to Compilers

Assignment 6: SEM1

Out: 10/10/03

Due: 10/17/03

1.0 Introduction

So far you should have built enough of the front end of your DECAF compiler so that you have can generate abstract syntax trees from valid DECAF programs. In this assignment we start semantic analysis.

As we have noted in class, semantic analysis of DECAF requires three passes over the abstract syntax trees. The first pass finds and defines classes, the second defines all the methods and fields, and the third defines local symbols and resolves names and expressions. In addition, we have noted that the trees produced initially need to be cleaned up a bit in order to facilitate later semantic analysis and code generation. This is typically done on the initial pass.

This assignment is going to involve implementing the first two passes over the abstract syntax tree. (The next assignment will involve the third pass.)

2.0 Requirements

The first pass should do at least the following:

- Link scopes to their parent scopes.
- Define all type names in the appropriate scope.
- Set the super class of each defined class.
- Set the associated scope of each defined class.
- Set the modifier and type fields for each field and local declaration, if needed.
- Add a default constructor to any class that has no constructor.
- Add a call to the super class constructor in any constructor that is lacking one.
- Add field initializations to all constructors after the call to super. Field initializations should look like `<field> = <LiteralInit>` where `<LiteralInit>` is the AST node `Asts::LiteralInitInfo`.

The second semantic pass is easier. All it needs to do is:

- Define each field in the appropriate scope.
- Define each method in the appropriate scope.

- If a method is not static, add ‘this’ as its first formal parameter.
- Set the type of each method to the appropriate method type.
- Ensure that constructors are not static.

You will also need to make a variety of semantic checks. These can be made on either pass and should at least (feel free to add more):

- Ensure that break and continue statements only occur inside loops.
- Ensure that the super statement only occurs as the first statement in a constructor.
- Ensure that no declarations occur in the then or else parts of if statements unless they are included in a block.
- Ensure that all returns in void methods (or constructors) have no associated values.
- Ensure that all returns in methods returning values have an associated value.
- Ensure that there is a return at the end of a method returning a value.
- Ensure that constructors are not declared static.

Note that the built-in definitions for Object, String, and IO have slightly modified abstract syntax trees, so you might have to take these into account during your passes. In particular, the methods in these classes have an EXTERNAL modifier set and have no method body in their abstract syntax tree.

3.0 Visitors

Your two passes should be implemented as subclasses of the class DecafSemanticsPass, which is a subclass of DecafVisitor. The default implementation of the visitXXX method for an AST node that is a direct subtype of DecafAst is to visit the children; the implementation of one that is not a direct subtype is to call the same method of the appropriate supertype. When you implement your subclass you might want to have the methods call the corresponding method in DecafVisitor. Note that all visit methods should be declared to throw a DecafException.

The subtype DecafSemanticPass provides an error method that takes an AST node and an error message and makes the appropriate call to the error handler.

4.0 Creating and Using Your Classes

You will need to modify your factory class so that the methods createSemanticTypePass and createSemanticMemberPass return a new instance of your visitor for pass 1 and pass 2, respectively.

5.0 Symbol Processing

You will also need an implementation of symbol processing (a subclass of `DecafSymProcess`, returned by the factory class `newSymbolProcessor`). We will provide you with a sample implementation (not guaranteed to be correct) that you can use as a starting point. Alternatively you can roll your own.

6.0 Looking Ahead

The next assignment, implementing the third semantic pass, is probably one of, if not the most complex in the course. You might want to devote some time this week, as you are getting used to thinking in terms of tree walks, to planning out how semantic resolution should be done. In particular, you might want to make notes as to what you think needs be done for each type of AST node in order to define locals, lookup all names, and check expression types. You should also read the semantics document so that you understand all the various nuances of type and symbol processing.