Overview

The second phase of the compiler performs semantic analysis of a source program. There are two primary goals in this phase:

- Process declarations, statements and expressions of an input program in order to construct a compact representation of the program. For your project, this representation will include symbol tables and parse trees.
- Check the input source program for semantic errors.

We have divided the semantic analysis phase into three parts. The first part will process declaration statements in order to build the various symbol tables. The second part will process statements and expressions to build the parse trees. Finally, the third part will use the symbol tables and the parse trees to perform semantic analysis.

Part 3.a: Processing of Declaration Statements

Due Date: May 6, 2011.

A Java^{--} program defines the following:

- Each file defines a set of classes.
- Each class defines a set of methods and a set of instance variables.
- Each method defines a set of local variables and a set of parameters.

In this assignment, start by building symbol tables to store information about classes, methods and variables of a program. For every input file, your compiler will construct a symbol table that will store information about the classes defined in the file. Similarly, for every class it will construct a symbol table that will store information about all variables and methods defined in the class.

In addition to the definitions of the individual symbol tables, your compiler will connect the different symbol tables on the basis of how program entities are structured. For instance, the symbol table for a class will contain entries for every method, where each method entry will be a pair (field name, symbol table for method).

Extend your parser so that it can process the declaration statements. Your program should output the symbol tables that it constructs during this phase. The output should capture the hierarchical nature of the definitions.

Part 3.b: Processing of Statements and Expressions

Due Date: May 13, 2011.

A Java^{--} program includes statements and expressions within method blocks. We construct parse trees to capture the essential elements of every statement and expression. For example, we capture, for every expression, the expression's various operators and operands, which can themselves be other expressions. So, for this part, you must construct parse trees and connect them to the symbol tables that you created in part 3 of the project.

Part 3.c: Semantic Analysis

Due Date: May 20, 2011.

The goal of this part is to collect and check the semantic attributes associated with the names used within a Java^{--} program. An implementation of this part requires the following:

- Traversing the symbol tables and parse trees.
- Gathering all semantic attributes (such as scope and type) as you traverse the trees and associate them with the names of a program. For instance, you can associate a type with each name during this traversal.
- Resolving all names: any time you come across an identifier during a traversal, make sure that you know where the identifier is defined. (You may even want to replace a reference to an identifier string in a parse tree by a reference to its symbol table entry.)
- Checking for semantic errors as you traverse the tree.

The following list summarizes the various semantic checks that you will need to employ:

- Ensure that variable names are unique within their scopes, i.e. a given name can only occur once in the current symbol table.

- Resolve declarations of variables with their usage and scope. Note that the resolution of the names is an important component of your semantic analysis phase, which you will need for later analysis. You should implement it first.
- Type check expressions and statements. This will require you to associate types with expressions and statements. You can implement it by associating types with the leaves (terminals of Java^{--}) of a parse tree and then by propagating the type information bottom up in the tree.
- Match method invocations with their declarations.
- Ensure that field access expressions are valid.
- Check valid usage of `this` and `super`.
- Check valid usage of `return` and `continue` statements.

For a complete treatment of all the semantic constraints to which you must adhere, please refer to the Java^{--} language specification document.

Error Detection and Recovery

In the syntactic analysis phase, your program stopped at the first appearance of lexical or parsing error. In this phase, you will extend the error detection and recovery capabilities so that more than one *semantic* error can be detected in a program. Your program can continue to exit at the first syntactic error. However, it must report all semantic errors in the program. You can do this by skipping all errors that may arise due to an earlier error. For instance, if you detect a semantic error that a variable `x` has not been defined, report an error and then skip all further errors that relate to `x`. Similarly, for any expression or the assignment statement, only initial type errors should be reported. If there is a type mismatch, or an invalid type specified in a sub-expression of an expression, stop the type checking for the expression enclosing the error. One way to do this is to add an ERRORTYPE to your set of types, and return that as the type of the sub-expression containing the error. Then, when ERRORTYPE is the type of one of the operands of an expression, the type checking can be stopped and ERRORTYPE again returned.

Output

The output of this phase will print out the symbol tables and the parse trees of correct programs. If there are semantic errors in an input program, print out all errors along with the line numbers where the semantic error occurred.

Hints

- You will need to modify your Yacc grammar by defining actions for grammar rules. It will involve identifying the symbol tables in which semantic information associated with a rule can be stored. Also, you will be creating parse tree nodes and connecting them together in order to build the tree.
- You will need to instantiate your symbol table package. In this phase, you can now start refining the attributes of the symbols. Still more refinement will take place during the semantic checking and the code generation phases.
- Much of the work in part 3.b is mechanical. That is, you will be creating classes for storing information about each non-terminal. I suggest that you use inheritance for minimizing the code that you need to write. I will hand out material on this separately.
- For the semantic analysis part, write a general routine for traversing your data structures. You can then use this routine for making multiple traversals.
- I have not put any constraints on the number of passes your semantic analyzer can make in order to perform semantic analysis. However, your job will be easier if you have a very clear sense of the various kinds of information you can gather in each phase.