Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 — Segment III: Semantic Analysis — Project Assignment — Tuesday, September 26

DUE: Wednesday, October 11

Warning: The static semantics phase of the project requires considerably more effort than the previous phases. Remember that it counts for 8% of your total grade.

Extend your compiler to find, report, and recover from semantic errors in Decaf/Espresso programs. Most semantic errors can be checked by testing the rules enumerated in the section "Semantic Rules" of Handout 6 for Decaf and Handout 7 for Espresso. These rules are repeated at the end of this handout as well. However, you should read Handout 6/Handout 7 in its entirety to make sure that your compiler catches all semantic errors implied by the language definition. We have attempted to provide a precise statement of the semantic rules in both handouts. If you feel that some semantic rule in the language definition may still have multiple interpretations, you should work with the interpretation that sounds most reasonable to you and clearly list your assumptions in your project documentation (you can also send email to `6.035-staff@mit.edu` asking for clarification).

This part of the project includes the following tasks[1]:

1. Add semantic actions (to your CUP parser specification) to construct a high-level intermediate representation (IR) tree, and to perform any other semantic checks that you choose to do on-the-fly as part of the parsing process.

   We will supply a basic IR tree framework: `java6035.tools.IR`. You can either build your tree on top of this framework (recommended), or write your own IR trees from scratch (in this case you should talk to your TA first). Our framework isn't complete; you'll still need to define classes for types and symbol tables. Our framework will also work with a visual debugging tool that can create a graphical representation of your program. More detailed descriptions of the IR framework are available in Handout 16 and in the online Javadoc documentation.

   When running in debug mode, your driver should print the constructed IR tree. A pretty-printing feature is implemented for you in our IR framework (`PPrint()` method of `Walkable` nodes). If you chose not to use our framework, you will need to implement a printing feature on your own.

2. Build symbol tables for the classes.

3. Construct and insert high-level IR code to perform the run-time checks listed in the language handout and repeated at the end of this handout.

4. Perform all remaining semantic checks by traversing the IR and accessing the symbol tables.

[1]See chapters 4 and 5 of the main textbook.

What to Hand In

Follow the directions given in Handout 4 when writing up your project. Your design documentation should include a description of how your IR and symbol data table structures are organized, as well as discussion of your design for performing the semantic checks.

The electronic copy portion of the hand-in procedure is similar to that of the parser segment. Provide a gzipped tar file named `leNN-semantics.tar.gz` in your group locker, where `NN` is your group number.. This file should contain all relevant source code and a `Makefile`. Additionally, you should provide a Java archive, produced with the `jar` tool, named `leNN-parser.jar` in the same directory.

Unpacking the tar file and running `make` should produce the same Java archive. With the `CLASSPATH` set to `leNN-parser.jar:/mit/6.035/classes`, you should be able to run your compiler from the command line with:

`java Compiler` <*filename*>

The resulting output to the terminal should be a report of all static errors encountered while compiling the file. Your compiler should give reasonable and specific error messages (with line numbers and identifier names) for all errors detected. It should avoid reporting multiple error messages for the same error, e.g., it `y` has not been declared in the assignment statement "`x=y+1;`", the compiler should report only one error message for `y`, rather than one error messages for `y`, another error message for the `+` and yet another error message for the assignment. After you implement the static semantic checker your compiler should be able to detect and report *all* static (i.e., compile-time) errors in any input Decaf/Espresso program, including lexical and syntax errors detected by previous phases of the compiler. In addition, your compiler should not report any messages for valid Decaf/Espresso programs. However, we do not expect you to avoid reporting spurious error messages that get triggered by error recovery – it is possible that your compiler might mistakenly report spurious semantic errors in some cases, depending on the effectiveness of your parser's syntactic error recovery.

In addition, your compiler should have a debug mode in which the IR and symbol table data structures constructed by your compiler are printed in some form. This can be run from the command line by

`java Compiler -debug` <*filename*>

Test Cases

The test cases provided for this project are several programs in Decaf/Espresso in

`/mit/6.035/provided/semantics/tests/`

Read the comments in the test cases to see what we expect your compiler to do.

If your compiler passes all of the test cases (accepts the semantically correct programs and rejects the incorrect ones) you get 10 points of the possible 30 points for implementation. The other 20 points will be based on how well your compiler performs on hidden test cases.

Semantic Rules for Decaf

1. No identifier is declared twice in the same scope.

2. No identifier is used before it is declared or after the block in which it is declared ends.
3. The program should contain a definition for a method called **main** that has no parameters (note that since execution starts at method **main**, any methods defined after main will never be executed).
4. The ⟨int_literal⟩ in an array declaration must be greater than 0.
5. The number and types of arguments in a method call must be the same as the number and types of the formals i.e., the signatures must be identical.
6. If a method call is used as an expression, the method must return a result.
7. A **return** statement must not have a return value unless it appears in the body of a method that is declared to return a value. If the method is declared to return a value, **return** statements in that method must have a return value of the specified type.
8. The expression in a **return** statement must have the result type of the enclosing method definition.
9. An ⟨id⟩ used as a ⟨location⟩ must name a declared local/global variable or formal parameter.
10. For all locations of the form ⟨id⟩`[`⟨expr⟩`]`
    (a) ⟨id⟩ must be an **array** variable, and
    (b) the type of ⟨expr⟩ must be **int**.
11. The ⟨expr⟩ in **if** and **while** statements must have type **bool**.
12. The operands of ⟨arith_op⟩s and ⟨rel_op⟩s must have type **int**.
13. The operands of ⟨eq_op⟩s must have the same type, either **int** or **bool**.
14. The ⟨location⟩ and the ⟨expr⟩ in an assignment, ⟨location⟩ `=` ⟨expr⟩, must have the same type.

Run Time Checks for Decaf

1. The subscript of an array must be in bounds.
2. Control must not fall off the end of a method that is declared to return a result.

Semantic Rules for Espresso

1. No identifier is declared twice in the same scope.
2. No identifier is used before it is declared (in an appropriate scope) or after the block in which it is declared ends.
3. The program should contain a definition for a class called **Program** with a method called **main** that has no parameters. (Note that since execution starts at method **main**, any methods defined after it will never be executed.)

4. The number of arguments in a method call must be the same as the number of formals, and the types of the arguments in a method call must be compatible with the types of the formals.
5. If a method call is used as an expression, the method must return a result.
6. A **return** statement must not have a return value unless it appears in the body of a method that is declared to return a value.
7. The expression in a **return** statement must have the result type that is compatible with the enclosing method definition.
8. For all method invocations of the form ⟨simple_expr⟩`.`⟨id⟩`()`
   (a) the type of ⟨simple_expr⟩ must be some class type, T, and
   (b) ⟨id⟩ must name one of T's method
9. An ⟨id⟩ used as a ⟨location⟩ must name a declared variable or field.
10. For all locations of the form ⟨simple_expr⟩`[`⟨expr⟩`]`
    (a) the type of ⟨simple_expr⟩ must be an instantiation of **array**, and
    (b) the type of ⟨expr⟩ must be **int**.
11. For all locations of the form ⟨simple_expr⟩`.`⟨id⟩
    (a) the type of ⟨simple_expr⟩ must be some class type, T, or ⟨simple_expr⟩ must be `this` (in which case its type is that of the enclosing class.)
    (b) ⟨id⟩ must name one of T's fields, and
    (c) the location must appear textually in an method of type T.
12. The ⟨expr⟩ in **if** and **while** statements must have type **boolean**.
13. The operands of ⟨arith_op⟩s and ⟨rel_op⟩s must have type **int**.
14. The operands of ⟨eq_op⟩s must have compatible types (i.e. either the first operand is compatible with the second or vice versa.).
15. The operands of ⟨cond_op⟩s must have type **boolean**.
16. In calls to **new** `T()`, T must be the name of a class previously defined in the global scope.
17. In calls to **new** `T[size]`, size must be an integer, and T must be either **int**, **boolean**, or the name of a class in the global scope. Note that T can *not* be an array type.
18. A field of a class T1 can only be accessed outside an operation for T1 if the caller is an operation in class T2 such that T2 is derived from T1.
19. The **extends** construct must name a previously-declared class; a class cannot extend itself.
20. The same name cannot be used for a field or operator in two separate class's when one class is an ancestor of the other in the inheritance graph.
21. The type of the ⟨expr⟩ in an assignment must be compatible with the type of the ⟨location⟩.
22. If T2 is derived from T1 and ⟨id⟩ is a method name defined in both T1 and T2, then m must have exactly the same signature and T1 and T2.

Run Time Checks for Espresso

1. The subscript of an array must be in bounds. If `a` is an array of size `N`, the index must be in the range term 0 ... (`a.length`-1).
2. In calls to **new** `T[size]`, size must be non-negative.
3. Control must not fall off the end of a method that is declared to return a result.
4. The value of a location must be non-**null** whenever it is dereferenced using `.` or [].