

**Massachusetts Institute of Technology**  
**Department of Electrical Engineering and Computer Science**

6.035, Fall 2005

Handout 8 — Semantic Analysis Project Wednesday, September 21

---

**DUE: Thursday, October 06**

**Warning: The static semantics phase of the project requires considerably more effort than the previous phase.**

Extend your compiler to find, report, and recover from semantic errors in Decaf programs. Most semantic errors can be checked by testing the rules enumerated in the section “Semantic Rules” of Handout 6. These rules are repeated at the end of this handout as well. However, you should read Handout 6 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. If you feel some semantic rule in the language definition may 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. Add semantic actions to your CUP parser specification to construct a high-level intermediate representation (IR) tree, and to perform any semantic checks you choose to do as part of the parsing process. The problem of designing an IR will be discussed in the lectures; some hints are given in the final section of this handout.

When running in debug mode, your driver should pretty-print the constructed IR tree in some easily-readable form suitable for debugging.

2. Build symbol tables for the classes. (A symbol table is an environment, i.e. a mapping from identifiers to semantic objects such as variable declarations. Environments are structured hierarchically, in parallel with source-level constructs, such as class-bodies, method-bodies, loop-bodies, etc.)
3. Perform all remaining semantic checks by traversing the IR and accessing the symbol tables.  
**Note:** the run-time checks are not required for this assignment.

## What to Hand In

Follow the directions given in Handout 3 when writing up your project. Your design documentation should include a description of how your IR and symbol table structures are organized, as well as a discussion of your design for performing the semantic checks.

The electronic portion of the hand-in procedure is similar to that of the previous segment. Provide a gzipped tar file named `leNN-semantic.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-semantic.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/provided/jars/java_cup.jar:...`, 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 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. For example, if `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 message 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 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 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.

Your semantic checker should verify that all decimal and hex integer literals in the input (which your scanner will have stored simply as strings) have numeric values within the range permitted by the specification in Handout 5.

As mentioned, 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 in:

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

Read the comments in the test cases to see what we expect your compiler to do. Points will be awarded based on how well your compiler performs on these and hidden tests cases. Complete documentation is also required for full credit.

## Semantic Rules

These rules place additional constraints on the set of valid Decaf programs besides the constraints implied by the grammar. A program that is grammatically well-formed and does not violate any of the following rules is called a *legal* program. A robust compiler will explicitly check each of these rules, and will generate an error message describing each violation it is able to find. A robust compiler will generate at least one error message for each illegal program, but will generate no errors for a legal program.

1. No identifier is declared twice in the same scope.
2. No identifier is used before it is declared.
3. The program contains 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  $\langle \text{int\_literal} \rangle$  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.
8. The expression in a **return** statement must have the same type as the declared result type of the enclosing method definition.
9. An  $\langle \text{id} \rangle$  used as a  $\langle \text{location} \rangle$  must name a declared local/global variable or formal parameter.
10. For all locations of the form  $\langle \text{id} \rangle [\langle \text{expr} \rangle]$ 
  - (a)  $\langle \text{id} \rangle$  must be an **array** variable, and
  - (b) the type of  $\langle \text{expr} \rangle$  must be **int**.
11. The  $\langle \text{expr} \rangle$  in **if** and **while** statements must have type **boolean**.
12. The operands of  $\langle \text{arith\_op} \rangle$ s and  $\langle \text{rel\_op} \rangle$ s must have type **int**.
13. The operands of  $\langle \text{eq\_op} \rangle$ s must have the same type, either **int** or **boolean**.
14. The operands of  $\langle \text{cond\_op} \rangle$ s and the operand of logical not (!) must have type **boolean**.
15. The  $\langle \text{location} \rangle$  and the  $\langle \text{expr} \rangle$  in an assignment,  $\langle \text{location} \rangle = \langle \text{expr} \rangle$ , must have the same type.
16. All **break** and **continue** statements must be contained within the body of a loop.

## Implementation Suggestions

- You will need to declare classes for each of the nodes in your IR. In many places, the hierarchy of IR node classes will resemble the language grammar. For example, a part of your inheritance tree might look like this (where indentation represents inheritance):

```

abstract class Ir
abstract class   IrExpression
abstract class   IrLiteral
    class         IrIntLiteral
    class         IrBooleanLiteral
    class         IrCallExpr
    class         IrMethodCallExpr

```

```

class          IrCalloutExpr
class          IrBinopExpr
abstract class IrStatement
class          IrAssignStmt
class          IrBreakStmt
class          IrContinueStmt
class          IrIfStmt
class          IrWhileStmt
class          IrReturnStmt
class          IrInvokeStmt
class          IrBlock
class          IrClassDecl
abstract class IrMemberDecl
class          IrMethodDecl
class          IrFieldDecl
.
.
.
class          IrVarDecl
class          IrType

```

Classes such as these implement the *abstract syntax tree* of the input program. In its simplest form, each class is simply a tuple of its subtrees, for example:

```

public class IrBinopExpr extends IrExpression
{
    private final int      operator;
    private final IrExpression lhs;
    private final IrExpression rhs;
}

```

|  
+  
/ \  
lhs rhs

or:

```

public class IrAssignStmt extends IrStatement
{
    private final IrLocation lhs;
    private final IrExpression rhs;
}

```

:=  
/ \  
lhs rhs

In addition, you'll need to define classes for the semantic entities of the program, which represent abstract properties (e.g. expression types, method signatures, class descriptors, etc.) and to establish the correspondences between them. Some examples: every expression has a type; every variable declaration introduces a variable; every block defines a scope. Many of these properties are derived by recursive traversals over the tree.

As far as possible, you should try to make instances of the the symbol classes *canonical*, so that they may be compared using reference equality. You are strongly advised to design these classes with care, employing good software-engineering practise and documenting them, as you will be living with them for the next few months!

- All error messages should be accompanied by the filename and line number of the token most relevant to the error message (use your judgement here). This means that, when building

your abstract-syntax tree (or AST), you must ensure that each IR node contains sufficient information for you to determine its line number at some later time.

(Some industrial compilers, e.g. the Jikes compiler for Java, associates with each IR node the *start* and *end* character positions of the first and last tokens included in that construct, allowing its error messages to report exactly which part of a source line contains an error. This can also be done easily with the JavaCUP framework, which exposes the left and right character positions of each grammar symbol *n* in the variables `left` and `right`.)

It is *not* appropriate to throw an exception when encountering an error in the input: doing so would lead to a design in which at most one error message is reported for each run of the compiler. A good front-end saves the user time by reporting multiple errors before stopping, allowing the programmer to make several corrections before having to restart compilation.

- Semantic checking should be done **top-down**. Recall that AST construction in LR-parsers is done bottom-up. While the type-checking component of semantic checking can be done in bottom-up fashion, other kinds of checks (for example, detecting uses of undefined variables) can not.

There are two ways of achieving this. The first is to make use of parser actions *in the middle of productions*, for example:

```
block ::= LBRACK                {: envs.push(new Env()); :}
        statements:s            {: checkStmts(s); :}
        RBRACK                  {: envs.pop(); :}
```

In this production, a new environment is created and pushed on the environment stack, the body of the block is then checked in the context of this environment stack, and then the new environment is discarded when the end of the block is reached. This approach requires less code but can be more complex, because it changes the production rules of the grammar in ways which can introduce conflicts.

A cleaner approach is to invoke your semantic checker on a complete AST after parsing has finished. The pseudocode for `block` in this approach would resemble this:

```
void checkBlock(EnvStack envs, Block b) {
    envs.push(new Env());
    foreach s in b.statements
        checkStatement(envs, s);
    envs.pop();
}
```

The semantic check can thus be expressed as a *visitor* (which should be familiar from *Design Patterns* or 6.170) over the AST. (Since code generation, certain optimizations, and pretty-printing can also be naturally expressed as visitors, we recommend this approach for a cleaner implementation.)

- The treatment of negative integer literals requires some care. Recall from the previous hand-out that negative integer literals are in fact two separate tokens: the positive integer literal and a preceding ‘-’. Whenever your top-down semantic checker finds a unary minus operator, it should check if its operand is a positive integer literal, and if so, replace the subtree (both nodes) by a single, negative integer literal.