Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Spring 2013 — Handout — Semantic Analysis Project — Wednesday, Feb 20

DUE: Monday, Mar 4 9:00 pm

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 the Decaf Spec. These rules are repeated in this handout as well. However, you should read the Decaf Spec 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-tas@mit.edu` asking for clarification).

This part of the project includes the following tasks:

1. Create a high-level intermediate representation (IR) tree. You can do this either by instructing ANTLR to create one for you, adding actions to your grammar to build a tree, or walking a generic ANTLR tree and building a tree on the way. 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 methods. (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 method-bodies, loop-bodies, etc.)

3. Perform all 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 project overview handout 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.

Each group will have their own private github repository. Students must take care to avoid accidentally (or intentionally) sharing their code with other students. Such actions will constitute cheating. The layout for directories and files in your repository (e.g. `build.sh`, `run.sh`) should be the same as it was for the scanner / parser project. You should also add an AUTHORS file in the top level of the project with your team members' names, one per line.

You should be able to run your compiler from the command line with:

```
./run.sh --target=inter <filename>
```

The resulting output to the terminal should be a report of all errors (printed to standard output) encountered while compiling the file. Your compiler should give reasonable and specific error messages (with line and column 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.

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 using the command:

```
./run.sh --target=inter --debug <filename>
```

Getting Started

Once you have finalized your group, the course staff will give you access to your team's repository. You may then clone the repository with the following command:

```
add -f git
cd ~/Private/6.035
git clone https://github.com/6035/<team repository name>.git
cd <team repository name>
```

You may then want to initialize your repository with one team member's code; you may do that by having that team member do the following (note that all team members must have pushed all changes from their first projcet to their first project repositories):

```
git remote add p1 https://github.com/6035/<team member's username>6035.git
git pull p1 master
# Fix the conflict in the readme and then
git commit -a
```

Finally, you should add the tests repository as a remote:

```
git remote add tests https://github.com/6035/tests.git
```

Test Cases

The test cases provided for this project have been added to the **tests** repository on Github. They can be added to your repo by running (as always)

```
  git pull tests master
```

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.

1 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. This includes **callout** identifiers, which exist in the global 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 ⟨int_literal⟩ in an array declaration must be greater than 0.
5. The number and types of arguments in a method call (non-callout) 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. String literals and array variables may not be used as arguments to non-callout methods. **Note: `a[5]` is not an array variable, it is an array location**
8. 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.
9. The expression in a **return** statement must have the same type as the declared result type of the enclosing method definition.
10. An ⟨id⟩ used as a ⟨location⟩ must name a declared local/global variable or formal parameter.
11. For all locations of the form ⟨id⟩`[`⟨expr⟩`]`

    (a) ⟨id⟩ must be an **array** variable, and

   (b) the type of ⟨expr⟩ must be **int**.

12. The ⟨expr⟩ in an **if** or a **while** statement 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 the same type, either **int** or **boolean**.
15. The operands of ⟨cond_op⟩s and the operand of logical not (**!**) must have type **boolean**.
16. The ⟨location⟩ and the ⟨expr⟩ in an assignment, ⟨location⟩ `=` ⟨expr⟩, must have the same type.
17. The ⟨location⟩ and the ⟨expr⟩ in an incrementing/decrementing assignment, ⟨location⟩ `+=` ⟨expr⟩ and ⟨location⟩ `-=` ⟨expr⟩, must be of type **int**.
18. The initial ⟨expr⟩ and the ending ⟨expr⟩ of **for** must have type **int**.
19. All **break** and **continue** statements must be contained within the body of a **for** or a **while**.

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         IrPlusAssignStmt
         class         IrBreakStmt
         class         IrContinueStmt
         class         IrIfStmt
         .
         .
         .
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, 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. In Scala, case classes handle this well. You are strongly advised to design these classes with care, employing good software-engineering practice and documenting them, as you will be living with them for the next few months!

- All error messages should be accompanied by the filename, line and column 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.

  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**. 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*. This approach may require less code but can be more complex, because more work needs to be done directly within the ANTLR infrastructure.

  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());
    for each statement in b.statements
        checkStatement(envs, statement);
    envs.pop();
}
```

In this pseudocode, 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.

The semantic check, just like code generation and certain optimizations, can often be expressed as a *visitor* (which should be familiar from *Design Patterns* or 6.005/6.170/?) over the AST. However, the visitor pattern may be most useful in imperative programming, and is just one design. For languages that support first-class functions, there are certainly many other interesting desgins. Whatever your language and design, we strongly recommend you separate your logic for performing generic AST operations (traversing, searching, etc) from the logic of your compiler functions (e.g. semantic checks). Think carefully about what designs will work well for performing many analyses and operations on an AST!

- One last note: the treatment of negative integer literals requires some care. Recall from the previous handout 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.