CS 321 Winter 2014 Homework 5: Static Analysis
Due (via D2L) on Friday, March 14, 2014 at noon

This assignment explores static analysis. Specifically, it asks you to implement analyses that detect *unreachable* statements in miniJava programs.

You will implement your analyses by writing code to traverse the miniJava AST trees that you targeted in Homework 4. The assignment assumes familiarity with the OO tree-traversal techniques discussed in Labs 7 and 8, and is intended to be attempted after you have completed those labs.

To complete this assignment, you will need a Java compiler and working installation of `javacc`. (If you are planning to use your own computer and have not yet installed `javacc` on it, see the instructions in Homework 3 for how to do so.)

You should download the "Homework 5 Materials" (`hw5.zip`) from the D2L website. Upon unzipping, you should see a `hw5` directory with the following items:

`ast`	a package containing file `Ast.java`, which contains AST node definitions (slightly changed from homework 4)
`ast0`	a package containing a modified version of `Ast.java` that illustrates tree traversal
`astParser.jj`	a parser for reading back the dumped AST format
`TestReachability.java`	a top-level test driver
`Makefile`	for building the parser and test driver
`tst`	a directory containing sample valid miniJava test programs and their corresponding AST dump outputs (unchanged from homework 4)

Your solution should be submitted using the D2L dropbox in the form of a single file `hw5.zip`, which in turn contains *three* files called `ast1/Ast.java`, `ast2/Ast.java`, and `exact.txt`. Assuming those three files are in the current directory and its package subdirectories, you can create a zip file containing them by typing

```
prompt> zip hw5.zip ast1/Ast.java ast2/Ast.java exact.txt
```

Further details on what should be in the three files are given below.

This assignment will be scored out of 55 points, following the distribution indicated below. Your score on this assignment contributes 15% towards your final grade. The minimum passing score (i.e. to avoid an F for this assignment, and for the class) is 10 points.

Detecting Unreachable Statements

As you may have observed, the `javac` compiler sometimes rejects a Java program because it contains an *unreachable statement*. A statement is unreachable if control can *never* pass to the beginning of that statement in *any* run of the program (no matter what values are given as input, if the program requires any). Since unreachable statements cannot affect the behavior of the program, it is reasonable to treat their presence as indicating a mistake on the programmer's part. (It is not uncommon for compilers for other languages to flag unreachable statements with a warning; Java is somewhat unusual in making them hard errors.)

`javac` performs a static analysis to detect and flag unreachable statements. Like many other static analyses, this one calculates an *approximation* of the program's runtime behavior; that is, its prediction of the runtime behavior may

be wrong in some cases. But the analysis only errs in one direction: it might fail to flag some statements that really cannot be reached, but it never flags a statement that might actually be reached during some program run. This means the compiler will never reject a program unless it is *certain* that it contains one or more unreachable statements; this behavior is appropriate because unreachability just indicates a likely mistake, not a fatal problem, so we don't want programs that only *might* contain unreachable statements to be rejected. (Notice that this is different from the way the compiler treats other, similar, static analyses, such as checking that the program is free from potential runtime type errors, or that variables are initialized before they are used; in these cases, the compiler rejects a program unless it is certain that they don't contain an error.)

In this assignment, you will write (several versions of) an unreachability analysis for the miniJava language introduced in Homeworks 3 and 4. (Recall that the syntax of miniJava is a strict subset of full Java.) You will implement the analysis by adding new methods to the miniJava AST, i.e. by modifying the `Ast.java` file. For this assignment, you need to start from a slightly different version of `Ast.java` than you used in Homework 4. The new starting version, which is in the Homework 5 materials on the D2L website, differs only in that: (i) the `Statement` class contains a new instance variable `public boolean reachable`, and (ii) whenever the AST dumping mechanism prints a statement, it puts an exclamation mark (`!`) in the first column of the line if the `reachable` flag for that statement is false.

Your task is to write code that correctly sets the `reachable` flag to `true` or `false` on every statement in a program. (The precise definition of "correct" is discussed below.) The top-level entry-point to your code must be a new instance method `void setReachability()` in the `Program` class.

To help you get started, the package subdirectory `ast0` contains a modified version of `ast/Ast.java` that contains a very simple (and highly incorrect!) version of the `setReachability()` method. Note that the `reachable` flag on every statement is initialized to `true` when the `Stmt` node is constructed. The implementation of `setReachability()` given here traverses the AST and sets the flag to `false` on every statement. The point of this is just to illustrate what a simple tree traversal looks like in the AST context. You will still need to change the behavior of this traversal (and perhaps add other traversal functions) to get the desired behavior for your analysis code.

To test your analysis code, you will need to try it out on the ASTs of various miniJava programs. File `astParser.jj` provides a parser for the AST dump format (`.ast`) files that you generated in Homework 4. It can be compiled using `javacc` in the usual way. The top-level driver program `TestReachability.java` operates by reading in the `.ast` file specified on the command line, thus generating an internal `Ast` tree; invoking `setReachability()` on the root of that tree; and finally redumping the tree (annotated with `!`'s as appropriate) to standard out.

Note that to run the code from `ast0` instead of `ast`, you'll need to change the `import ast.*` statements at the top of `TestReachability.java` and `AstParser.jj` to import `ast0` instead of `ast`, and recompile everything.

To obtain suitable `.ast` files for testing, you can either run your own parser from Homework 4, or else write programs directly in the `.ast` dump format. The latter is not at all difficult to do, especially if you use an existing `.ast` file as a template; many such files can be found in the `tst` subdirectory. Note that few if any of these existing test files contain unreachable statements. It is your responsibility (and a major part of your task) to come up with appropriate tests. It is permissible to share test files with other students.

You are asked to write two different versions of the reachability analyzer. Analyzer 2 refines and builds on Analyzer 1, so you want to get a complete working version of 1 before starting on 2. You are also asked to write about (but not code) a potential third version of the Analyzer.

As an example, suppose we have a (mini)Java source file `test.java`:

```
class test {
  public static void main(String[] a) {
    return;
    System.out.println("hello");
  }
}
```

with corresponding dump file `test.ast`:

```
# AST Program
 ClassDecl test
  MethodDecl void main ()
   Return ()
   Print "hello"
```

Then running either version of your analyzer should produce the following:

```
prompt> java TestReachability test.ast
# AST Program
 ClassDecl test
  MethodDecl void main ()
   Return ()
!  Print "hello"
```

1. [25 pts] For the first version of the analysis, make the simplifying assumption that the values of boolean expressions are *always* unknown, no matter how trivial they might be. For example, even in a statement like

```
if (true)
  x = 0;
else
  x = 1
```

your analyzer should not try to figure out which branch of the `if` is taken (and hence will not be able to flag `x = 1` as unreachable). You should also assume that the first statement of every function is reachable. Otherwise, your analysis should be as precise as possible.

In grading this problem, primary emphasis will be placed on behavioral correctness of your code, i.e., how well it does on the tests *we* devise. However, if you wish to obtain partial credit for less-than-perfect solutions, you should try to make your code as intelligible as possible, and to include comments that explain your overall strategy.

How you organize your code is up to you, but you are strongly encouraged to implement a version of the `setReachability` method in the `Statement` class, defined separately for each kind of statement using the OO style described in Lab 7. You may also find it very useful to define additional other, auxiliary methods that compute useful attributes of nodes.

Put your modified version of `Ast.java` into a new package `ast1`. To do this, create a new subdirectory `ast1`, place your modified `Ast.java` in this subdirectory, and modify the `package` declaration at the top of the file from `ast` or `ast0` to `ast1`. (As with `ast0`, you'll need to change the `import` statements in `AstParser.jj` and `TestReachability.java` in order to use your new package.) Include the file `ast1/Ast.java` in the zipfile that you submit to D2L.

You will wish to have a fully working version of this analyzer before moving to part 2.

2. [25 pts] The second version of your analyzer should determine the values of *constant expressions* of type `boolean` or `integer`. An expression is constant if and only if it is:

- a boolean literal (i.e `true` and `false`);
- an integer literal; or
- an arithmetic, boolean, or relational operator applied to two constant expressions.

Then, your analyzer should use its information about constant expressions to improve the accuracy of its reachability results. For example, in analyzing the fragment

```
while (2 + 2 == 4) {};
x = 1
```

the analyzer should discover that the test expression in the `while` statement evaluates to the constant `true`, and that therefore the `while` statement loops forever, and hence that the statement `x = 1` is unreachable. But the analyzer should continue to learn *nothing* about the value of other kinds of expressions; for example, in

```
if (a == a)
  x = 0;
else x = 1
```

it still should not know which branch of the `if` is taken (and hence should not discover that `x = 1` is unreachable).

In coding this analysis, you may assume that the AST has already been typechecked, i.e. that it represents a type-correct program. Type correctness in miniJava is essentially the same as in Java. Warning: Note that the equality (`EQ`) and inequality (`NE`) operators can be validly applied to a pair of operands of *any* type, including object types—but you should only treat the result as a constant expression when the operands are `boolean` or `integer` constant expressions.

Put your modified version of `Ast.java` into a new package `ast2`, analogously to `ast1` above. (Once again, change `import` statements as necessary to use package `ast2` in place of `ast1`.) Include the file `ast2/Ast.java` in the zipfile that you submit to D2L.

Hints:

- Add an instance method `cval()` to the `Expression` class that returns its value (as an `Integer` or `Boolean` object) if it is a constant expression, and `null` otherwise.
- In designing test miniJava programs, you may find it useful to try `javac` on them to see if it detects unreachable statements. But be warned that `javac`'s analysis is an odd mixture of the two analysis variants described here: more precisely, it uses the second variant everywhere *except* when analyzing the test expressions of `if` statements, where it uses the first variant. (Can you think why this might be?)

3. [5 pts] Is it possible to write an *exact* static analysis for unreachability for miniJava, i.e. one that always flags exactly the truly unreachable statements? Give a brief, informal, but precise response that either:

(a) describes how to implement such an analysis; or

(b) explains why such an analysis cannot exist.

(It may be useful to note that miniJava lacks any mechanism for performing input.)

Put your explanation in a *plain text* file called `exact.txt` and include that file in the zip file that you sumbit to D2L. (Please do *not* submit files in other formats such as `.doc`, `.rtf`, or `.pdf`.)