Lab 9: Static Analysis

CS 320 Principles of Programming Languages, Winter Term 2020
Department of Computer Science, Portland State University

Learning Objectives

Upon successful completion, students will be able to:

- Investigate and debug implementations of common static analyses.
- Review key components of the phase structure of a typical compiler and the role of each phase.
- Use object-oriented programming techniques, particularly inheritance and dynamic dispatch, to implement computations on ASTs.

Instructions

Download the file `lab9.zip` from D2L into any convenient directory and type

```
$ unzip lab9.zip
```

You'll see a new directory `lab9`. Unlike the previous labs, this one does not have numbered sub-directories. All the work and exercises pertain to the single directory `lab9` and its named sub-directories. You are asked to complete four exercises, which will lead to the creation of some new files and the modification of others.

When you have completed all the exercises, return to the top-level directory above `lab9` and type

```
$ zip -r sol9.zip lab9/
```

Then submit the resulting file `sol9.zip` to D2L.

Introduction

This set of lab notes provides an introduction to the use and implementation of static analysis in a simple compiler for a programming language that we will refer to as "`mini`". This language is rather basic, lacking many of the features that you might expect to find in a more realistic programming language. Nevertheless, it is still rich enough to illustrate key concepts for CS 320.

Programs in `mini` look syntactically a lot like fragments of Java programs, and they have broadly similar semantics. But don't get confused: `mini` is a different language! In parciular, `mini` is not a strict subset of Java, and you cannot compile or run `mini` programs using `javac` and `java`.

This set of notes is packaged together with the full source code for the `mini` compiler. (There are a few bugs in the code that you will be asked to fix as an exercise.) You are encouraged to take a look at any parts of the code that are of interest to you, but you should also note that detailed understanding of this code base, or of exactly how each part of it works, is beyond the scope of this course.

The `mini` language is an example of an imperative programming language, meaning that `mini` programs are written as sequences of commands or statements, each of which performs a specific action—such as updating the value of a variable or printing out a value—when the program is executed. The following listing shows a `mini` program for calculating the sum of the numbers from 1 to 10 and printing out the result:

"test1.mini"

```
1  int i, t;
2  i = t = 0;
3  while (i<10) {
4    i = i + 1;
5    t = t + i;
6  }
7  print t;
```

This program uses two variables: a counter, `i`, that takes values from 0 to 10; and a total, `t`, that is used to accumulate the sum of all the different values of `i`. If you have any doubt about what this program does or how it works, you should try performing a dry run, tracing the values of the two variables at each step. If you conclude that the program should terminate after printing out a final result of 55, then you are in good shape to carry on!

Should you need more information about the syntax of `mini`, you can find a detailed grammar in the file `grammar.txt` that is included with these notes. That file, for example, can be used to determine the relative precedence and fixities for each of the operators in the `mini` language. (Quick quiz: For implementation reasons, one of the most common operators has actually been left out of the grammar; can you figure out what that might be? And which has lower precedence between the `&&` and the `&` operators?)

Building and running the `mini` compiler

You can use the following commands to build and run the `mini` compiler:

- `make` will attempt to rebuild the `mini` compiler after you have made changes to its source code.
- `make clean` will remove all of the temporary files (`.class` files, for example).
- `java MiniCompiler file` will run the `mini` compiler with a source file called `file.mini` (you should *NOT* include the "`.mini`" part of the name in the command line that you use). If successful, this will create an assembly language version of the source program in `file.s` and an executable version of the source program in `file`.

The following output shows a sequence of commands that can be used to build the `mini` compiler; use it to compile the `test1` sample program; and then to run the generated program:

```
$ make
rm -f test?.s buggy?.s test? buggy?
rm -f MiniCompiler.class */*.class
```

```
javac MiniCompiler.java

$ java MiniCompiler test1
Assembly code output: test1.s
Executable program: test1

$ ./test1
print: 55

$
```

The code that is produced by this version of the `mini` compiler is only intended to be used on departmental Linux machines. You may be able to get the code to compile on other platforms by adjusting the `platform` variable in `ast/IA32Output.java`, and by ensuring that you have an appropriate set of IA32 (32-bit) tools available, but you are own your own. You can, however, build and run the `mini` compiler itself on any platform that has a suitable version of Java installed; you just won't be able to compile and run the generated assembly code outputs.

Inside the `mini` compiler

The `mini` compiler is implemented by code in a small number of packages: `compiler` (general utilities for error handling); `lexer` (lexical analysis); `parser` (parsing); and `ast` (abstract syntax tree classes). All of the code that we will be working with in this lab is in the `ast` package, or in the `MiniCompiler.java` source file that appears in the top-level folder. The compiler uses a standard structure with components that are organized into:

- a front end, which mostly handles syntax analysis;
- a middle end, which mostly handles semantic analysis; and
- a back end, which handles optimization and code generation.

This structure is reflected in the following three lines of the main function in `MiniCompiler.java` (the other parts of that function have to do with handling command line arguments and exceptions, in much the same style as we have seen in previous labs):

"MiniCompiler.java"

```
14  Stmt prog = frontEnd(handler, args[0] + ".mini");
15  middleEnd(args[0], handler, prog);
16  backEnd(args[0], prog);
```

In this case, a complete program, `prog`, is represented as a single statement (in fact, as a `Block` statement that contains multiple statements in sequence).

Our focus in this lab is on the middle end, which is implemented by the following suitably named method in `MiniCompiler.java`:

"MiniCompiler.java"

```
38  // Analyze program:
39  private static void middleEnd(String name, Handler handler, Stmt prog) throws Exception {
40    // new TextOutput(System.out).toText(prog);
```

```
41      // new IndentOutput(System.out).indent(prog);
42
43      // String dot = name + "_ast.dot";
44      // new DotOutput(dot).toDot(prog);
45      // System.out.println("AST after parsing: " + dot);
46
47      new ScopeAnalysis(handler).analyze(prog);
48
49      // String html = name + ".html";
50      // new HTMLOutput(html).toHTML(prog);
51      // System.out.println("Resolved program: " + html);
52
53      // String env = name + "_env.dot";
54      // new DotEnvOutput(env).dotEnv(prog);
55      // System.out.println("Environment graph output: " + env);
56
57      new TypeAnalysis(handler).analyze(prog);
58
59      new InitAnalysis(handler).analyze(prog);
60
61      //  new UseAnalysis(handler).analyze(prog);
62    }
```

The commented-out lines in this method produce various kinds of diagnostic information; their use will be described further below. Each of the three remaining lines represents a different kind of static analysis:

- Scope analysis, which checks that variables are only used when they are in scope;
- Type analysis, which checks to make sure that all expressions in the program have correct/compatible types; and
- Initialization analysis, which checks to make sure that all variables are initialized before they are used.

You can comment out one or more of these lines too, to disable specific forms of analysis. However, it is important to understand that each of these different analyses is written with the assumption that the preceding analyses have been run successfully (i.e., without detecting any errors). For example, there is no point in trying to find the type of an expression (during type analysis) if that expression mentions an undefined variable (that should have been detected during scope analysis).

Looking at ASTs

The front end of our compiler will construct an abstract syntax tree for the `mini` source program that is passed as input. We have seen similar code in previous labs for lexing and parsing a simple expression language. For the purposes of this lab, we can think of the front end as a utility function that turns text files into trees so that we can focus on writing programs that perform computations on those trees.

There are several different ways that we can modify the code for the `middleEnd()` function that will help us to visualize the AST structures that are produced by the front end:

- In simple text format, by uncommenting the following line:

"MiniCompiler.java"
```
40      // new TextOutput(System.out).toText(prog);
```

(This is sometimes called "pretty printing" because it attempts to display the program in a properly indented format, even if the original was not correctly formatted. Then again, this is somewhat subjective, and this version of the printer does not show comments, and also inserts some redundant parentheses. As such, perhaps the output will not seem quite so "pretty" for some readers ...)

- Using indentation to show tree structure, by uncommenting the following line:

"MiniCompiler.java"
```
41    // new IndentOutput(System.out).indent(prog);
```

This produces plain text output that shows the tree structure of an AST by ensuring that the children of each node are indented to the right of the node itself. This can produce some long outputs, but clearly captures the structure of the generated tree.

- Using dot to construct a graphical diagram of the tree structure, by uncommenting these lines:

"MiniCompiler.java"
```
43    // String dot = name + "_ast.dot";
44    // new DotOutput(dot).toDot(prog);
45    // System.out.println("AST after parsing: " + dot);
```

With these lines in place, each (successful) run of the `mini` compiler on file `foo.mini` will create a file called `foo_ast.dot` in the current working directory. The contents of that file can then be turned into a graphical version of the tree by using a command like the following:

```
dot -Tpdf -o foo_ast.pdf foo_ast.dot
```

Alternatively, you can view the graph generated by `foo_ast.dot` directly by using the web viewers at `www.webgraphviz.com` or `dreampuf.github.io/GraphvizOnline`.

In the dot output, statement nodes are colored `"lightblue"` and expression nodes are colored `"salmon"`. If you move these lines to after the TypeAnalysis line, then the colors of the expression nodes will change to reflects the type of the value produced by that node: `"lemonchiffon"`=int, `"plum"`=boolean.

Class Hierarchy Diagrams

Even though `mini` is quite a simple language, we still need quite a lot of different classes to describe its AST structures. To help understand and work with large sets of classes like this, it can be useful to see those classes arranged in a hierarchy that captures important details—such as the inheritance relationships between classes and the names of fields—in a compact notation.

The "`hierarchy.pdf`" file that is included with these materials shows two distinct abstract syntax tree hierarchies:

- one for statements, rooted in the `Stmt` class; and
- one for expressions, rooted in the `Expr` class.

Here are some guidelines on how to read hierarchy diagrams like this:

- Each box in the diagram corresponds to a class in the `ast` package.

- The name of each class is written (in blue) on the left side of the corresponding box.
- The diagram shows the hierarchy of the classes, with superclasses on the left and the associated tree of subclasses (inheriting from their superclass(es)) on the right.
- Abstract classes just have a name (and possibly a list of one or more field names that are present in all subclasses).
- Classes representing specific language construct have a name and, on the right side, a representative fragment of concrete syntax to show what the associated construct might look like in a real program:
- The variable names in the concrete syntax examples are chosen to match the corresponding field names in the classes.
- Subscripts are used to identify fields that are implemented as arrays (such as the body of a `Block` or the vars in a `VarDecl`).

The class hierarchy plays a critical role in structuring the code:

- Operations that must be implemented for every different form of expression, for example, are specified by inserting an appropriate abstract method definition in the `Expr` superclass.
- When a given field or method should be included in all subclasses of a given class, we can take advantage of inheritance, inserting a single definition instead of having to write essentially the same code in multiple places.

In short, finding the right place to insert the definition of a new field or method in a class hierarchy can be an important step in minimizing coding effort and code duplication.

Now we will start looking in more depth at each of the three different forms of static analysis that are used in the `mini` compiler.

Scope Analysis

The purpose of scope analysis is to track which variables are in scope at each point of the program. This meets two specific needs in the `mini` compiler:

1. To detect references to variables that have not been previously declared so that they can be reported to the user as program errors.
2. To determine suitable locations in memory for storing each variable. Because this has to do with details of the `mini` compiler's back end that are outside the scope of this lab, we will not say anything further here about this aspect of scope analysis.

Scope analysis is implemented by traversing the abstract syntax tree of the source program. At each point, the analysis maintains an "environment", which is a mapping that lists all of the variables that are in scope, including both the name and type of each variable. Concretely, an environment is implemented using a linked list. This allows declarations of a variable with a given name to shadow declarations for the same name in an outer scope. It also makes it easy to share the common parts of different environments: for example, although the statements in the two branches of an `if` statement could potentially have different environments, they will share a common tail, corresponding to the environment immediately before (and after) the `if` statement.

The following code fragment shows a simplified version of the interface/implementation for environments:

```
/** Represents an environment that stores information about the
 *  type of each variable in a program.
 */
public class Env {
    private Id   id;    // The identifier for this environment entry.
    private Type type;  // The type for this environment entry.
    private Env  rest;  // Enclosing items for this environment entry.

    public Env(Id id, Type type, Env rest) {
        this.id = id;
        this.type = type;
        this.rest = rest;
    }

    public static Env lookup(Id id, Env env);
    public Id   getId();
    public Type getType();
}
```

To implement scope analysis, we need to visit every statement and expression in the abstract syntax for the program that we are analyzing. (If we do not visit some part of the tree, then we may not find the definition of a variable, or we may overlook a use that does not have a corresponding definition.)

For expressions:

"ast/Expr.java"

```
70  /** Run scope analysis on this expression.  The scoping parameter
71   *  provides access to the scope analysis phase (in particular,
72   *  to the associated error handler), and the env parameter
73   *  reflects the environment in which the expression is evaluated.
74   *  Unlike scope analysis for statements, there is no return
75   *  result here: an expression cannot introduce new variables in
76   *  to a program, so the final environment will always be the same
77   *  as the initial environment.
78   */
79  public abstract void analyze(ScopeAnalysis scoping, Env env);
```

Note that there are several methods called `analyze` in `Expr.java` and in many of the other classes. Java can tell them apart because they have different argument types; this is called *method overloading*. For example, the first argument of the scope analysis analyzer is a `ScopeAnalysis` object.

There are implementations of this scope analysis method in `Id.java`, `IntLit.java`, `BoolLit.java`, `Assignment.java`, `UnExpr.java`, and `BinExpr.java` (all in the `ast` package) that we should look at to get a better understanding of how this works. We'll look at `Id`, `IntLit` and `BinExpr` as examples; note that the latter provides an implementation that works for *all* binary operators, so we do not have to provide individual implementations for each one.

For statements, there is an added possibility that the environment may change from one statement to the next as the result of a variable declaration:

"ast/Stmt.java"

```
68  /** Run scope analysis on this statement.  The scoping parameter
69   *  provides access to the scope analysis phase (in particular,
70   *  to the associated error handler), and the env parameter
71   *  reflects the environment at the start of the statement.  The
```

```
72      *  return result is the environment at the end of the statement.
73      */
74    public abstract Env analyze(ScopeAnalysis scoping, Env env);
```

There are implementations of this method in `Block.java`, `Empty.java`, `ExprStmt.java`, `If.java`, `Print.java`, `VarDecl.java`, and `While.java` (again, all in the `ast` package/folder).

In addition to what has been described above, each environment node holds a list, called `uses`, that tracks the list of places where that variable has been used in the program. Every time we perform a successful environment lookup for a given variable name (`Id`), we add a new entry to the `uses` list, and we also set the `v` field of the `Id` to point to the environment. These extra details, capturing the relationship between definitions and uses of variables, can provide new insights about a program's structure that were not visible in any previous steps of the compilation process. In lexical analysis and parsing, for example, we may encounter two distinct occurrences of the same identifier, but we would not have any way to determine whether they refer to the same item or not. With the additional details collected here, we can determine whether two `Id`s refer to the same variable by checking to see if they have the same `v` field.

There is an additional component of the `mini` compiler sources that generates HTML versions of a given abstract syntax tree; this can be accessed by uncommenting the following lines of code:

"MiniCompiler.java"
```
49    // String html = name + ".html";
50    // new HTMLOutput(html).toHTML(prog);
51    // System.out.println("Resolved program: " + html);
```

If you open the generated html file in a browser, you will get a web page that changes dynamically:

- to highlight the "definition" of a variable every time you move the mouse to hover over a "use";
- to highlight the "uses" of a variable every time you hover over a "definition".

Again, this kind of functionality is only made possible as a result of information that is collected during scope analyis.

A further way to examine the results of scope analysis is to uncomment the next group of lines:

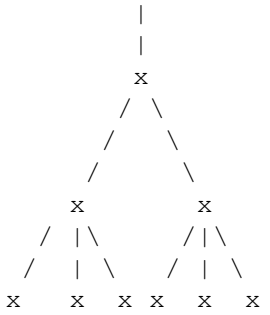
"MiniCompiler.java"
```
53    // String env = name + "_env.dot";
54    // new DotEnvOutput(env).dotEnv(prog);
55    // System.out.println("Environment graph output: " + env);
```

This produces another dot file, this one containing a graph of the environment data structures built by the scope analysis. Each node in the graph corresponds to a variable declaration and points to the next outermost declaration.

Exercise 1 [5 pts]:

Write a file `shape.mini` that, assuming you have uncommented the lines described just above, generates a file `shape_env.dot` which contains an environment graph with the following shape:

x



Here each `x` can be any identifier.

Leave your file in the top-level `lab9/` directory so that it will be included when you zip that directory for submission.

Type Analysis

The purpose of type analysis is to check that every part of a program has a valid type. This meets some specific needs in the full compiler:

1. To detect type inconsistencies in a program, such as an attempt to apply an arithmetic operation to a boolean, to use an integer as a test in a while loop, or to use a boolean as the argument to a print statement (which, in `mini`, can only be used to print integers).
2. To provide typing information that can be used in the back of the compiler to guide code generation. For example, it turns out that our compiler must generate different code for a bitwise not operation (the unary `~` operator, corresponding to the `BNot` class), depending on whether the argument is a boolean or an integer. Further details of this, however, are beyond the scope of this lab.

Type analysis is again implemented by traversing the abstract syntax tree of the source program. The interface to type analysis for expressions is defined by the following:

"ast/Expr.java"

```
83  /** Run type checking analysis on this expression.  The typing
84   *  parameter provides access to the type analysis phase (in
85   *  particular, to the associated error handler).
86   */
87  public abstract Type analyze(TypeAnalysis typing);
```

(Note that Java's method overloading mechanism can distinguish this method from the scope analysis analyze method we looked at before because its first argument is an object of type `TypeAnalysis`.) In the `mini` language, there are only two types, int and boolean, and these are represented by the constants `Type.INT` and `Type.BOOLEAN`, and each call to the analyze function above should return one of these two values as its result.

There are corresponding implementations of this method in each of the following files (all in the `ast` folder): `Id.java`, `IntLit.java`, `BoolLit.java`, `Assignment.java`, `BNot.java`, `LNot.java`, `UnArithExpr.java`, `BinArithExpr.java`, `BinBitwiseExpr.java`, `BinCompExpr.java`, `BinEqualityExpr.java`, and `BinLogicExpr.java`. (Rather than read through and remember this full list, it may be easier for you to identify the places in the class hierarchy diagram where implementations are required. For example, it

should be clear that `Add`, `Mul`, and `Sub` all have the same type checking rules, so we can cover all of those cases by adding a suitable implementation in the `BinArithExpr` class.

One important additional requirement of type analysis is to store the type of each expression in the `type` field of the corresponding abstract syntax tree node. This, for example, is why the implementation for type checking the code for an `LNot` construct (corresponding to the `!` operator) is written as follows, including an assignment to capture the result that it returns in the associated `type` field:

"ast/LNot.java"

```
29  public Type analyze(TypeAnalysis typing) {
30      return type = exp.require(typing, Type.BOOLEAN);
31  }
```

This code example also illustrates a use of the `require()` utility method, whose purpose is to check that a given expression has the expected type:

"ast/Expr.java"

```
89  /** Run the type checking analysis on an expression that is required to
90   *  have the type specified by the expected parameter.
91   */
92  Type require(TypeAnalysis typing, Type expected) {
93    Type t = analyze(typing);
94    if (t!=expected) {
95      typing.report(new Failure(pos, "An expression of type " + expected +
96                                     " was expected"));
97      return expected;
98    }
99    return t;
100 }
```

Our `mini` compiler must also include code for type checking statements, with the primary responsibility of ensuring that the type analysis visits every expression that is nested within a statement somewhere in the source program. This portion of the interface to the type checker is described by the following definition:

"ast/Stmt.java"

```
81  /** Run type checker on this statement.  The typing parameter
82   *  provides access to the type analysis phase (specifically,
83   *  to the associated error handler).
84   */
85  public abstract void analyze(TypeAnalysis typing);
```

Note that this version of `analyze` is defined as a `void` method: statements are executed in order to carry out some action, but they do not return any type of result. The implementations of this method may be found in the following classes (all in the `ast` folder): `Block.java`, `Empty.java`, `ExprStmt.java`, `If.java`, `Print.java`, `VarDecl.java`, and `While.java`.

Exercise 2 [10 pts]:

Write a file `bad.mini` that generates as many different type error messages as possible. No two of the messages you generate may be the same. For full credit, your `mini` code should provoke at least five different messages. Note that type errors are different from scope errors or initialization errors; your program must abort during type analysis, not during scope or initializaion analysis.

Leave your file in the top-level `lab9/` directory so that it will be included when you zip that directory for submission.

Hint: You can address this from first principles (what kinds of errors can there be?) and/or by investigating the code added to the various classes of `Ast` for method `analyze(TypeAnalysis typing)`.

Initialization Analysis

The purpose of initialization analysis is to ensure that all variables are initialized before they are used. This meets two specific needs in the `mini` compiler:

1. Using an uninitialized variable may given unpredictable results, so it is useful instead to detect and report such situations as an error.
2. If we know that all program variables will be properly initialized by user code, then there is no need for the compiler to generate additional code to initialize them. This can help to reduce the size and performance of the generated code in modest, but perhaps useful ways. Again, further details are beyond the scope of this lab.

Initialization analysis is implemented by adding implementations of the following method at appropriate points in the inheritance hierarchy for expressions and statements. (The signature shown here is for expressions; the one for statements, defined in `ast/Stmt.java`, is identical.)

"ast/Expr.java"

```
116   /** Run initialization analysis on this expression.  The init parameter
117    *  provides access to the initialization analysis phase (in particular,
118    *  to the associated error handler), and the initialized parameter
119    *  reflects the set of variables that are known to have been initialized
120    *  before this expression is evaluated.  The return result is the set of
121    *  variables that are known to be initialized after the expression has
122    *  been evaluated.
123    */
124   public abstract VarSet analyze(InitAnalysis init, VarSet initialized);
```

This method involves a new type, `VarSet`, which is used to represent sets of variables. The intention here is that:

- The value of the `initialized` parameter contains the set of variables that are guaranteed to have been initialized before the associated statement/expression is executed/evaluated; and
- The result is the set of variables that have are guaranteed to have been initialized after the associated statement/expression has been executed/evaluated.

Two key parts of the implementation of initialization analysis are in the code for checking variable references and assignments. In the first case, we include code to check that the `initialized` set contains the variable that is being referenced, and report an error if it does not. Either way, the set of variables that are guaranteed to have been initialized is not changed as a result of refering to a variable, so the same `initialized` set is also returned as a result:

"ast/Id.java"

```
189   public VarSet analyze(InitAnalysis init, VarSet initialized) {
190       if (!VarSet.includes(v, initialized)) {
```

```
191             init.report(new Failure(pos,
192                                     "The variable \"" + this
193                                     + "\" may be used before it has been initialized"));
194         }
195         return initialized;
196     }
```

Variables are initialized when they are used on the left hand side of an assignment statement, after the right hand side value has been evaluated. As a result, the set of variables that are guaranteed to have been initialized at the end of an assignment is just: the set of variables that had been initialized previously, plus the set of variables that are initialized in the right hand side, plus the variable that appears on the left of the assignment.

"ast/Assignment.java"

```
91  public VarSet analyze(InitAnalysis init, VarSet initialized) {
92      return lhs.addTo(rhs.analyze(init, initialized));
93  }
```

You can find additional implementations of initialization analysis:

- for expressions: in the `Id.java`, `IntLit.java`, `BoolLit.java`, `Assignment.java`, `UnExpr.java`, and `BinExpr.java` classes.
- for statements: in the `Block.java`, `Empty.java`, `ExprStmt.java`, `If.java`, `Print.java`, `VarDecl.java`, and `While.java` classes.

Exercise 3 [10 pts]:

It turns out that the implementation of initalization analysis in the supplied version of the `mini` compiler is not quite correct!

The problem is illustrated by the following short program:

"buggy3.mini"

```
1  boolean x, y;
2  if ((x=true) || (y=true))
3    if (y) print 0; else print 1;
```

The supplied version of the `mini` compiler accepts this program without complaint. But this program should actually be rejected because it will use the variable `y` before it has been properly initialized. The default assumption in the current initialization analysis is that the arguments of any binary operator (i.e., of any construct represented by a subclass of `BinExpr`) will both be evaluated, moving from left to right. But this is not valid for the "shortcircuiting" logical operators `&&` and `||`. Each of these operators always evaluates its left argument, but only evaluates the right argument if necessary to compute the result. In particular, `&&` only evaluates its right argument if the left argument evaluates to true, and `||` only evaluates its right argument if its left argument evaluates to false.

Your task for this exercise is to add an appropriate implementation of the initialization analysis method to the supplied compiler to ensure that it calculates the correct `VarSet` output for any expression, including those that use `&&` or `||`. This just requires changing the code of one method in one file in `lab9/ast/`. Be sure that you give the correct behavior for the (non-buggy) file `good3.mini` as well as all other programs.

Leave your modified file in place so that it will be included when you zip `lab9/` and its sub-directories for submission.

Other Forms of Static Analysis

There are many other forms of static analysis that are used in the implementations of compilers, interpreters, and other language processing systems. The kinds of analysis that are required and/or useful can vary widely depending on the details of the language and the specific tool that is being implemented. The final exercise below suggests one such example that might be useful for the `mini` compiler.

Exercise 4 [15 pts]:

Scope analysis attempts to detect uses of variables that have not been declared. But what about variables that are declared but not used? For example, consider the following program:

"buggy4.mini"

```
1  int x;
2  print 12;
```

Although this program is technically valid, the definition of a variable `x` in the first line is pointless because the variable is never used. Some language designers might choose to treat programs like this as an error:

1. In a larger program, an unused variable like this might be an indication of an error in program logic: perhaps the programmer meant to use the variable `x` at some point, but accidentally used a different (incorrect) variable instead (maybe because they didn't realize that the original definition of `x` was shadowed by a locally defined variable with the same name).
2. Given that the variable is not used, the program could be simplified (and its memory consumption marginally reduced) by eliminating the variable declaration . . . without changing the overall program semantics.

Your task for this exercise is to implement a static analysis that can detect and report instances of unused variables like this. To be more precise, you should report any variable that has a declaration but never appears in any expression. (This is a conservative approximation to actual non-use: a variable might appear in an expression that never actually gets evaluated, but this analysis will still consider it as "used". For example, the analysis should not complain about the program `good4.mini`.)

This analysis task can be accomplished by a traversal of the abstract syntax tree for the statements in a program (i.e., by defining and implementing a suitable method in the hierarchy for `Stmt`). It is not necessary to consider expressions in this analysis. The key details that you need to complete the implementation can be found by inspecting the vars arrays in each `VarDecl` node that you find and by using a combination of the `getEnv()` and `getUses()` methods to identify the list of "uses" for each one.

A skeleton implementation of a `UseAnalysis` pass is already defined in `ast/UseAnalysis.java`. Currently its `analyze` method does nothing; you should fill in its body appropriately, taking the existing passes as models. You can invoke the pass by uncommenting this line

"MiniCompiler.java"

```
61  //  new UseAnalysis(handler).analyze(prog);
```

Leave your modified file in place so that it will be included when you zip `lab9/` and its sub-directories for submission.