CS321 Homework 4 — Sample Solution

November 2012

Question 1: When we run the supplied grammar through `jacc`, a single conflict is reported:

```
$ jacc -v mini/Mini.jacc
WARNING: conflicts: 1 shift/reduce, 0 reduce/reduce
$
```

Notice that I have used the `-v` flag here, which tells `jacc` that we want to produce debugging output in the file `mini/Mini.output`. (We could also have used the `-h` flag to produce a corresponding HTML version of the debugging output.) Opening up the output file in an editor, we can search for the string "`conflict`", which brings us to the following state (with some details elided for brevity):

```
105: shift/reduce conflict (shift 116 and red'n 22) on ELSE
state 105 (entry on stmt)
    stmt : IF test stmt_ELSE stmt    (21)
    stmt : IF test stmt_    (22)

    ELSE shift 116
    ...
```

Now we can recognize this as an instance of the classic "dangling else" problem, with a shift/reduce conflict that occurs when the parser handles a statement of the following form:

```
IF (e1) s1 ELSE IF (e2) s2_ELSE s3
```

At the point where the parser reaches the position indicated by the underscore, there are two choices: either *reduce* the `IF (e2) s2` portion using the rule 22 mentioned in the output, or *shift* the `ELSE` token. As we can see from the line `ELSE shift 116` at the bottom of the description of State 105 above, the `jacc`-generated parser chooses the second option, shifting to State 116. As a result, the parser will end up reducing `IF (e2) s2 ELSE s3` as a single unit, and so the example above will be parsed as if it were written:

```
IF (e1) s1 ELSE { IF (e2) s2 ELSE s3 }
```

This is the expected way to parse this statement in common languages like C, Java, etc., so we can conclude that the shift/reduce conflict is being resolved in an appropriate way, and that there is no need to worry about the conflict warning message.

The next part of the question mentions an "unusual" precedence declaration for `[` as a right grouping operator. Sure enough, we can find the declaration `%right '['` on Line 29 of the supplied `mini/Mini.jacc` file, which is immediately after all of the other fixity declarations. Commenting that line out and rerunning jacc now reports a total of 22 shift/reduce conflicts:

```
$ jacc -v mini/Mini.jacc
WARNING: conflicts: 22 shift/reduce, 0 reduce/reduce
$
```

We know already that one of these can be explained by the dangling else problem described previously. For the remainder, we need to look for conflict reports in the new version of `mini/Mini.output`:

- The first thing we can notice is that **four** of the conflicts arise in connection with productions involving unary operators (there are four such operators in Mini: unary plus and minus, as well as the logical and bitwise negation operators). For example, the following shows the description of State 78 in abbreviated form:

```
78: shift/reduce conflict (shift 62 and red'n 34) on '['
state 78 (entry on expr)
  expr : '!' expr_    (34)
  expr : expr_'[' expr ']' '=' expr    (53)
  expr : expr_'[' expr ']'    (54)
```

  The parser will encounter this state while parsing an expression of the form `!e1[e2]` just before the `[` character. In this case, a *reduce* step would interpret the expression as `(!e1)[e2]`, whereas a *shift* step would treat it as `!(e1[e2])`. The latter option gives array indexing a higher precedence than the unary operator, which is a common choice, and hence it is ok to use the `jacc`-generated parser with its default action to *shift*.

- Next, we find that there are **sixteen** conflict warnings associated with productions involving infix operators (Mini has a total of sixteen different infix operators: `EQL`, `GTE`, `LAND`, `LOR`, `LTE`, `NEQ`, `&`, `*`, `+`, `-`, `/`, `<`, `>`, `^`, `|`, and `=`). For example, a conflict arises in State 83 in connection with the `EQL` operator:

```
83: shift/reduce conflict (shift 62 and red'n 45) on '['
state 83 (entry on expr)
  expr : expr EQL expr_    (45)
  expr : expr_'[' expr ']' '=' expr    (53)
  expr : expr_'[' expr ']'    (54)
```

  The parser will encounter this state while parsing an expression of the form `e1 == e2[e3]`, again just before the `[` character. In this case, a *reduce* step would interpret the expression as `(e1==e2)[e2]` (which is clearly not what we want because a `boolean` value should not be used as an array), but the default action—a *shift* step—will treat it as `e1 == (e2[e3])`, which is entirely reasonable (and again gives array indexing a higher precedence than the binary operator).

- Finally, there is **one** additional conflict in State 123:

```
123: shift/reduce conflict (shift 62 and red'n 53) on '['
state 123 (entry on expr)
    expr : expr_'[' expr ']' '=' expr    (53)
    expr : expr '[' expr ']' '=' expr_    (53)
    expr : expr_'[' expr ']'    (54)
```

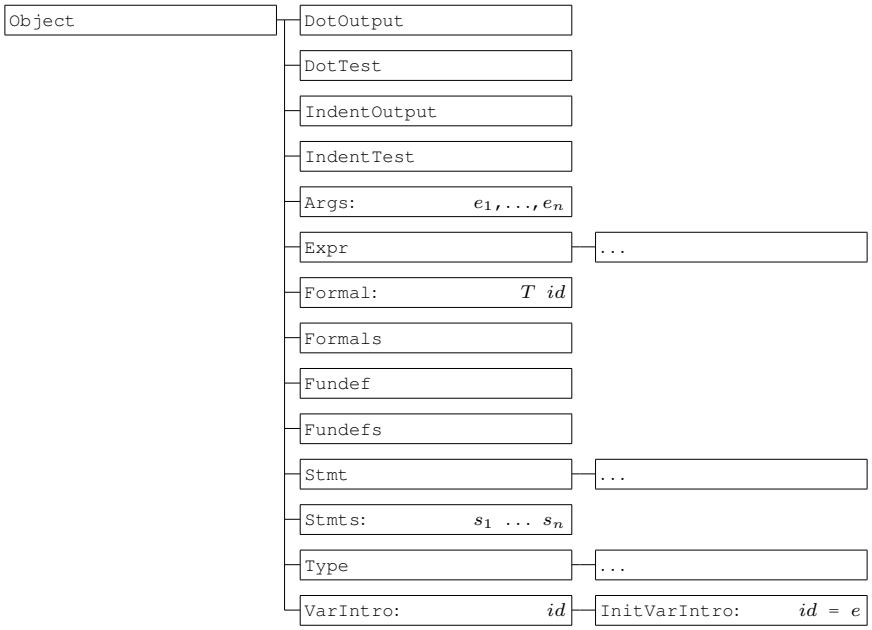
  The parser will enter this state while parsing an expression such as `a[i] = b[j]` immediately before the second `[` token. Once again, the correct (and `jacc`'s default) action here is to *shift*, which has the effect of treating the full expression as `a[i] = (b[j])`, as expected, instead of performing a *reduce* that would treat the input as `(a[i]=b)[j]`, which is unlikely to be what the programmer would expect. Note that the precedence of grammar rule 53 here is the same as the precedence `=`, which is lower than the precedence of `[`, so shifting corresponds to treating `[` as having a higher precedence.

We have now accounted for all 22 of the reported shift/reduce conflicts. In each case, the most appropriate way to resolve the conflict is by using a *shift* step, which is the default that is used by `jacc`. When we re-insert the original `%right '['` declaration, all but the original dangling else conflict message goes away. But we can still be sure that each of the 21 conflicts has been resolved appropriately because we put the fixity declaration for `[` at the end of the list so that it would have the highest precedence.

Question 2: The main purpose of this question is to gain a better understanding of the classes in the `mini` subdirectory that make up the abstract syntax for Mini. Using a command like `ls mini/*.java | wc -l`, we can count that there are 61 java source files in that folder (even more if you have already run `jacc`, `jflex`, or both), so this task might seem quite daunting. But all we really need to do if we're focussing on inheritance relationships is to determine which classes extend which, and that only requires us to look at the start of every class declaration. In particular, we can use a command like the following to extra the parts of the Java source files that describe the inheritance relationships:

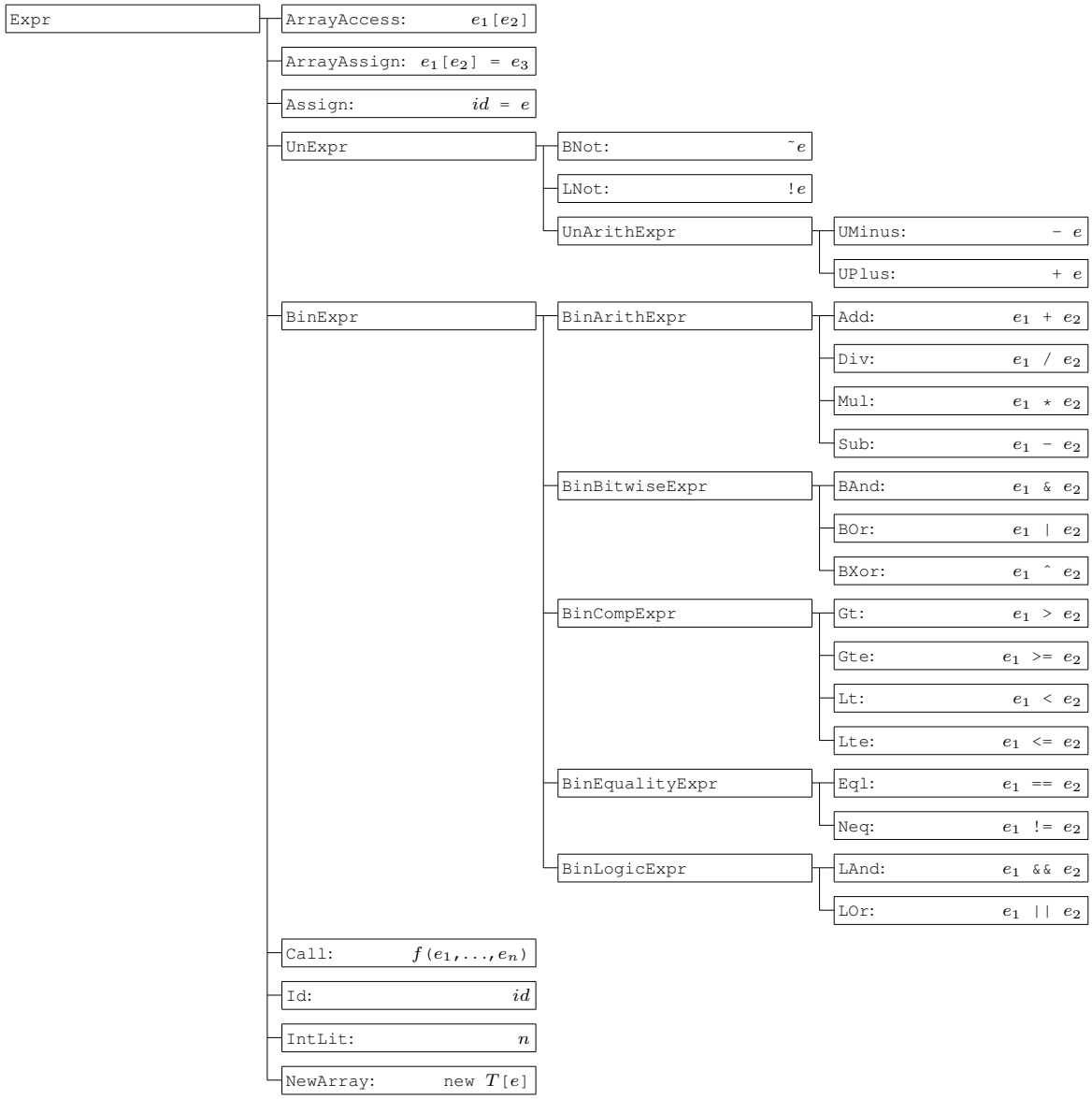
```
$ grep class mini/*.java
mini/Add.java:class Add extends BinArithExpr {
mini/Args.java:class Args {
mini/ArrayAccess.java:class ArrayAccess extends Expr {
...
mini/InitVarIntro.java:public class InitVarIntro extends VarIntro {
...
mini/VarIntro.java:public class VarIntro {
mini/While.java:class While extends Stmt {
$
```

Now we can start constructing a diagram that captures this information. For some classes (like `Args` in the fragment of output shown above), there is no explicit base class, so the built-in Object type serves as a base. There are 14 such classes in total, which we can document using the following diagram:



This includes four classes (`DotOutput`, `DotTest`, `IndentOutput`, and `IndentTest`) that are used for testing purposes; three classes (`Expr`, `Stmt`, and `Type`) that serve as the base classes for the abstract syntax of expressions, statements, and types, respectively; and a smattering of additional classes for representing other parts of the abstract syntax such as an argument list (`Args`), a formal parameter (`Formal`), a formal parameter list (`Formals`), a list of statements (`Stmts`), a function definition (`Fundef`), and a list of function definitions (`Fundefs`). In the diagram here, we label classes representing abstract syntax with not just the class name, but also a short fragment of concrete syntax to further illustrate what the class represents. We also include some boxes filled only with ... to suggest where the extensions of abstract classes will be placed.

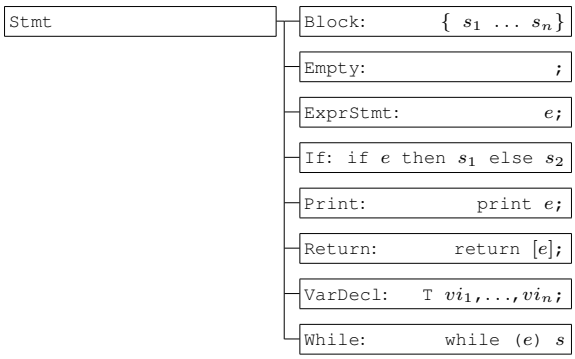
Finally, at the bottom of the diagram, we include the classes `VarIntro` and `InitVarIntro` that are used to represent the elements of a variable declaration that introduce a new variable name, either without, or with, a suitable initializer expression. As can be seen in the output from the initial `grep` command, the definition of `InitVarIntro` specifies that it `extends` another class, in this case, `VarIntro`. Once we start to focus on classes that either directly or indirectly extend the `Expr` class, we end up with the following diagram to show the hierarchy for classes that can be used in the representation of expressions:



This structure includes some additional abstract classes like `UnExpr` (a base class for all unary expressions), `BinExpr` (a base class for all binary expressions), and `BinArithExpr` (a base class for expressions using one of the four main arithmetic operators). This structure makes a lot of sense because it allows us to organize the classes in a helpful way, and to avoid duplicating common code by placing a unique copy in the appropriate class. For example, all binary expressions have both a left and a right argument, so the code for defining those fields will fit nicely in to the `BinExpr` class. In a similar way, all four of the main arithmetic operators use the same type checking rule, so we will be able to place the code for that in the

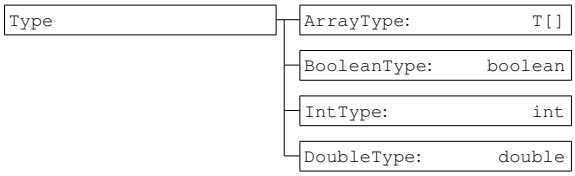
`BinArithExpr` class where it will be inherited by `Add`, `Div`, `Mul`, and `Sub`.

Now we can continue this process to construct a diagram that shows the relationships between the abstract syntax classes for statements:

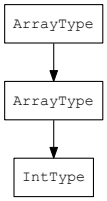


Note that the vi_1,...,vi_n values listed in the box for the `VarDecl` class stand for variable introductions, which are just instances of the `VarIntro` and `InitVarIntro` classes described previously.

Finally, we can construct a similar diagram for the classes that provide an abstract syntax for types:



It is important to realize that the diagrams shown here capture the inheritance relationships between the classes that are used to describe the abstract syntax of Mini, but they are not abstract syntax tree diagrams. For emphasize this point, the following diagram shows the abstract syntax tree for the Mini type `int[][]` (an array of arrays of integers), which could also be constructed directly in Java using the expression `new ArrayType(new ArrayType(new IntType()))`.



Question 3: The remaining task is to extend the Mini parser to support parsing of C/Java-stye for loops.

Part a) Our first challenge is to define some new classes for capturing the abstract syntax of `for` loops. Clearly, this will require the definition of a `For` class that `extends Stmt` and includes fields to store all four of the components of a `for` loop that are listed in the question. A natural way to begin constructing this code is to take the code for a similar, and already existing class (such as `While`), and then adapt it to fit the needs of `For`. In addition to the `For` class itself, we are going to need to find ways to represent the abstract syntax of a `for` loop initializer (which can be either empty, an expression, or a variable declaration), and the abstract syntax of the test and step parts (which can be either empty or an expression). In the following answer, I will demonstrate two different approaches to this: 1) We will define a small collection of classes to represent each of the different forms of initializer; in particular, we will represent an empty initializer by

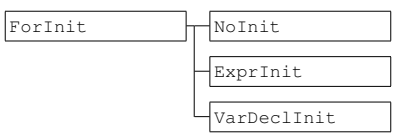
an object of type `NoInit`. 2) We will represent the test and step portions of a for loop using values of type `Expr`, using a `null` pointer rather than an object. As we will see, the first approach requires a bit more work up-front to define extra classes. However, the second approach requires special case treatment to deal with (potentially) `null` pointers.

The following code for the `For` class defines the four fields that represent the components (`init`, `test`, `step`, and `body`) of a `for` loop, a corresponding constructor, and an implementation of the indent method that is required in all subclasses of `Stmt`. As mentioned previously, we need to make special tests for `null` pointers in the parts of `indent` having to do with for `test` and `step` because invoking a method like `test.indent(out, n+1)` when `test` is `null` would cause an error at runtime.

```
class For extends Stmt {
    private ForInit init;
    private Expr    test;
    private Expr    step;
    private Stmt    body;

    For(ForInit init, Expr test, Expr step, Stmt body) {
        this.init = init;
        this.test = test;
        this.step = step;
        this.body = body;
    }

    public void indent(IndentOutput out, int n) {
        out.indent(n, "For");
        init.indent(out, n+1);
        if (test==null) {
            out.indent(n+1, "No test");
        } else {
            test.indent(out, n+1);
        }
        if (step==null) {
            out.indent(n+1, "No step");
        } else {
            step.indent(out, n+1);
        }
        body.indent(out, n+1);
    }

    public int toDot(DotOutput dot, int n) {
        return dot.node("- not implemented -", n);
    }
 }
```

(Note that I've used the stub definition for `toDot()` that was provided in the question. A proper implementation for this function is included in the accompanying code package.)

To account for the possibility that the initializer in a `for` loop could be either empty, an expression, or a variable declaration, we will define a small hierarchy of classes, as described in the following diagram:



The `ForInit` class shown here serves as an abstract base class and captures the fact that we want to be able to use `indent()` on each of the different forms of identifier:

```
public abstract class ForInit {
    public abstract void indent(IndentOutput out, int n);
}
```

The case of an empty initializer is represented by the `NoInit` class, which has no fields and just a simple implementation of `indent()`:

```
public class NoInit extends ForInit {
    public void indent(IndentOutput out, int n) {
        out.indent(n, "NoInit");
    }
}
```

The `ExprInit` class is used to represent expression initializers, so it only has a single `Expr` field:

```
public class ExprInit extends ForInit {
    private Expr expr;
    ExprInit(Expr expr) {
        this.expr = expr;
    }

    public void indent(IndentOutput out, int n) {
        out.indent(n, "ExprInit");
        expr.indent(out, n+1);
    }
}
```

In a similar way, `VarDeclInit` represents variable declaration initializers and holds a single `VarDecl`:

```
public class VarDeclInit extends ForInit {
    private VarDecl decl;
    VarDeclInit(VarDecl decl) {
        this.decl = decl;
    }

    public void indent(IndentOutput out, int n) {
        out.indent(n, "VarDeclInit");
        decl.indent(out, n+1);
    }
}
```

Part b) The next step is to modify the parser to recognize the syntax of `for` loops. First, we need to add a token to represent the `for` keyword. This requires an extra line in `mini/Mini.jflex` to ensure that the lexer will recognize the new token (I put this with the other lines that define keyword tokens):

```
"for"           { return FOR; }
```

We also need to add a declaration for `FOR` in the initial portion of the parser definition in `mini/Mini.jacc`:

```
%token  FOR
```

The biggest changes that we need to make are in the grammar. Even there, however, we only need to give one production for `for` loops, adding that to the others for the `stmt` (statement) nonterminal (the numbers here will be used to reference the productions later in the text):

```
stmt    : ...
        | FOR '(' init ';' optExpr ';' optExpr ')' stmt  { $$ = new For($3, $5, $7, $9); } // 0
        ;
optExpr : expr                                           { $$ = $1; }                      // 1
        | /* empty */                                    { $$ = null; }                    // 2
        ;
init    : varDecl                                        { $$ = new VarDeclInit($1); }     // 3
        | expr                                           { $$ = new ExprInit($1); }        // 4
        | /* empty */                                    { $$ = new NoInit(); }            // 5
        ;
```

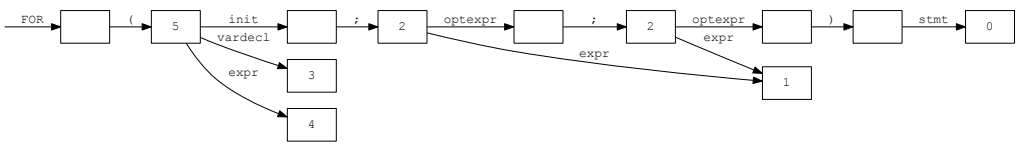
Notice that we have also defined nonterminals `optExpr` (for optional expressions in the `test` and `step` fields) and `init` (for initializers). By using these auxiliary nonterminals, we avoid having to write out eight separate variants of the syntax for `for` loops, each distinguished from the others by a different combination of the `init`, `test`, and `step` components. Notice also how the actions for the `optExpr` and `init` nonterminals match up with the representation choices that we made in Part (a): an optional expression is either a `null` or an expression; and every initializer is constructed using one of the three classes `NoInit`, `ExprInit`, or `VarDeclInit`. To reflect these choices, we must also make one more change in the top portion of `mini/Mini.jacc` to ensure that `optExpr` and `init` are associated with the appropriate classes:

```
%type   <Expr>    optExpr
%type   <ForInit> init
```

Now, at last, we can use `jflex` to generate the lexer, use `jacc` to generate the parser, and then use `javac` to compile all of the Java code:

```
$ jflex mini/Mini.jflex
....
$ jacc -v mini/Mini.jacc
WARNING: conflicts: 1 shift/reduce, 0 reduce/reduce
$ javac mini/*.java
$ tail -2 mini/Mini.output
67 grammar rules, 136 states;
1 shift/reduce and 0 reduce/reduce conflicts reported.
$
```

The output from the last command shows that the generated LR machine now has 136 states, which is **an increase of 12** over the original (as mentioned in lectures, but you can also verify this for yourself by running `jacc` on an unmodified copy of `mini/Mini.jacc`) These new states are needed, of course, to account for the new productions in the grammar. The full machine is obviously quite complicated, but if we just think about the states that are needed to parse a `for` loop, then we can see that the machine must include (at least) one fragment with the following structure:



Note that the numbers shown in some of the states correspond to reduce items for the productions with the same number in the grammar that was given previously. For example, the main horizontal sequence shown here corresponds to the states that the machine will follow as it advances through the symbols on the right of Production 0. It is easy to see that there are 12 states here, none of which could have been included in the original machine (because they lead to reduce items for productions that were not included in that grammar). As such, this accounts for all of the new states that were added to the LR machine as it went from a total of 124 to 136 states. Of course, integrating these new states in to the full machine requires the addition of extra transitions. For example, any state in the original machine with a goto on `stmt` will need the addition of a transition on `FOR` to the start state of the fragment of the machine shown above. In a similar way, the states in the above diagram that have goto transitions will also need the addition of shift transitions (for each symbol in the associated FIRST set) to other states in the original LR machine, as well as additional goto transitions that correspond to taking the closure of the item set in each state.

Part c) After all the work of extending the abstract syntax and the parser, it's time to do some testing. To get things started, it's a good idea to run some quick tests. For example, when I wrap the Mini code in the bottom right-hand corner below inside a function definition and run the resulting file through `java mini.IndentTest`, I get the output shown here (rearranged for the purposes of this presentation in to three columns):

```
For                    For                For
  VarDeclInit            NoInit             NoInit
    VarDecl              Lt, <              No test
      int                  Id, i            No step
      InitVarIntro         IntLit: 10       ExprStmt
        Id, i              No step            Assign
        IntLit: 0          Empty                Id, j
  Lt, <                                         Add, +
    Id, i                                         Id, j
    IntLit: 10                                    IntLit: 1
  Assign               ---------------------------------------
    Id, i              |  for (int i=0; i < 10; i = i+1)
    Add, +             |    print i;
      Id, i            |  for (; i < 10; )
      IntLit: 1        |    ;
  Print                |  for (; ; )
    Id, i              |    j = j+1;
```

These examples here were picked to show a quick cross-section of the `for` loop syntax; by checking the indented output it is easy to check that each of the three examples is displayed in the way that we would expect. In particular, we can check that every `For` node has exactly four subtrees; in the case when one (or more) of these components is omitted from the input syntax, we can see that a placeholder such as `NoInit`, `No test`, or `Empty` is shown in the indented output.

Simple examples like these are a good starting point for testing but it is hard to ensure comprehensive testing: there at least 8 different variations of the for loop depending on which of the init, test, and step fields are included, so we are clearly not getting good coverage with only 3 tests! However, it is also not easy to construct a large number of test cases by hand and to be sure that we have covered a broad range of cases. One way to deal with this here is to write a program that can generate a set of test cases. Popular scripting languages are good candidates for doing this, but I'll stick with Java here because that's the language I'm using in the rest of the class:

```
public class maketests {
  public static void main(String[] args) {
    String[] inits = new String[] {            // 7 options here
```

```
        "int i=0", "int i", "int i=0, j=1",
        "i=0", "i=j+1", "f(x,y)", ""
      };
    String[] tests = new String[] {            // 5 options here
        "i < 10", "i<10 && j>i", "b", "!b", ""
      };
    String[] steps = new String[] {            // 4 options here
        "i = i+1", "step", "func()", ""
      };
    String[] bodies = new String[] {           // 4 options here
        "print i;", ";", "{ print i; print j; }", "i = j+1;"
      };

    System.out.println("void fortests() {");
    for (int i=0; i<inits.length; i++) {
      for (int t=0; t<tests.length; t++) {
        for (int s=0; s<steps.length; s++) {
          for (int b=0; b<bodies.length; b++) {
            System.out.println("    for (" + inits[i] + "; "
                                           + tests[t] + "; "
                                           + steps[s] + ")");
            System.out.println("      " + bodies[b]);
            System.out.println();
          }
        }
      }
    }
    System.out.println("}");
  }
}
```

Running this program will print out a Mini test program with a single function definition that contains $7 \times 5 \times 4 \times 4 = 560$ different `for` loops. It does this by selecting options from lists that provide different options for filling in the components of a `for` loop, each of which includes the possibility of an empty item.

```
$ javac maketests.java
$ java maketests > for.mini
$ java mini.IndentTest for.mini
Fundef void fortests(..)
  Formals
  Stmts
    For
      VarDeclInit
        VarDecl
          int
          InitVarIntro
            Id, i
            IntLit: 0
      Lt, <
... about 8000 more lines ...
$
```

Although we could carry out spot checks on individual examples, it is hard to verify all of the output that is produced here by hand. But the fact that `IndentTest` terminates without an error indicates that our parser has accepted all of the different program variants that were constructed by `maketests`. With

this approach, we can be fairly confident that our testing has broad coverage. And because the process is automated, the testing process is quite fast too. Furthermore, if we later discovered that some particular test case had not been captured, it would likely be easy to extend `maketests` to include that case as well, rerunning it to produce a new version of the `for.mini` test file.

In fact, even though it is hard to automate the process of checking the output from the above program, there are some other tests that it would be potentially useful to automate. For example, if we arranged for all of the programs generated by `maketests` to be stored in separate files, then we could generate a corresponding set of indented output files. If all of the inputs are distinct, then all of the outputs should also be distinct (we could check for this possibility using a duplicate finder). A test procedure like this is useful because it can catch errors resulting from copy-and-paste mistakes. For example, if the constructor for `For` had an error because it assigned `this.step = test`, then we would discover that some of the output trees are displayed in the same way because the correct value of `this.test` was not stored in the tree.

In summary, in this assignment, we've seen and used several examples of programs that are used to produce other programs: we've used `jflex` to generate a lexer, `jacc` to generate a parser, and now `maketests` to generate test cases. The next time you are working on a programming assignment, it might be worth taking a brief pause to wonder "is there a program that I could use to write this code for me?"