Language Processors Lab 6

Note: Read this through *before* logging in.

The goal for this lab is to see the MiniJava typechecker in operation.

MiniJava typechecker

The MiniJava symbol table builder and typechecker is at `/soi/sw/courses/daveb/IN2009/minijava/chap5`. `README` files in each directory explain the basic structure and how to compile and run it. Move to where you want to do your work and copy it with:

```
cp -R /soi/sw/courses/daveb/IN2009/minijava/chap5 .
```

Move to your copy of the `chap5` directory and copy the contents of the directory containing Minijava sample programs:

```
cp -R /soi/sw/courses/daveb/IN2009/minijava/programs .
```

Compile with:

```
module add java soi javacc/3.2
javacc minijava.jj
javac Main.java
```

and run with:

```
java Main filename
```

where `filename` is a file containing a MiniJava program (eg one of those from the `minijava/programs` directory):

```
java Main programs/BinarySearch.java
```

The output is a pretty-printed version of the program, and any error messages resulting from the typechecking.

The programs supplied in directory `programs` contain no type errors so you won't see any error messages from them. You should now create some of your own test program files that contain simple errors that should be detected by the symbol table builder and type checker, and try them out. For example, you could make a copy of `BinarySearch.java` and introduce type errors.

Then read the slides from Session 5 and the visitor programs and try to understand how the symbol table is built and how the typechecker works.

Extending MiniJava and the Type Checker

In the previous lab (Lab 5) you were asked to introduce the `''/''` operator. In this section we will go through the steps required to extend the typechecker for the new `Division` expression.

- Extend MiniJava with the division operator. Follow the steps listed in lab5 (though you should know this by now). Note that the abstract syntax tree accepts a second visitor (`TypeVisitor`, look at the other classes in `syntaxtree`). Thus, the `Division` class implementation should look like this:

  ```
  package syntaxtree;
  import visitor.Visitor;
  import visitor.TypeVisitor;

  public class Division extends Exp {
    public Exp e1,e2;
  ```

```
  public Division(Exp ae1, Exp ae2) {
    e1=ae1; e2=ae2;
  }

  public void accept(Visitor v) {
    v.visit(this);
  }

  public Type accept(TypeVisitor v) {
    return v.visit(this);
  }
}
```

You will need to modify (as explained in lab5) the `minijava.jj`, `visitor/Visitor.java` and `visitor/DBPrettyVisitor.java` files.

Recompilation (compile the program again) should report an error. This is because we have not extended the typechecker visitor interface and implementation yet. The three visitors used by the typechecker and symbol table builder are subclasses of a general visitor - `TypeDepthFirstVisitor` - which implements the `TypeVisitor` interface. We should modify these files first.

- Modify the default visitor by adding:

  ```
          public Type visit(Division n);
  ```

  to the `visitor/TypeVisitor.java` file. Next, add a default `visit` implementation to the `TypeDepthFirstVisitor.java` file:

  ```
   public Type visit(Division n) {
     // Same as visit(Times n)
    }
  ```

  Compilation should give you no errors. However, this default implementation will not perform typechecking but will make sure that the whole tree is traversed. For example, modify the `return 0` at the end of the `Init` method in `BinarySearch.java` to `return 0 * true` (first) and `return 0 / true`, and compare the messages sent by the type checker. The first case will be correctly typechecked (since it's already implemented) but, in the second case, the checker will not notice that `true` is of the wrong type. The next step will implement correct typechecking for `Division`.

- Add `visit(Division n)` to `TypeCheckExpVisitor.java`. The implementation of this method should be similar to the one used for `Times`. Don't forget to display the right messages!