Compiler Project, Stage 2: Typechecking I

Computer Science 371
Amherst College
Spring 2014

This assignment, due on **Friday, February 21**, focuses on the first part of typechecking.

1 Getting Ready

Go into your `cs371` directory, and issue the following commands:

```
cp -r hw1 hw2
cp -r ~lamcgeoch/cs371/hw2/* hw2
[chmod -R a+w hw2]     # do this, without the brackets, only if you're in a group directo
cd hw2
rm Makefile
ln -s Makefile2 Makefile
make
```

The effect of these commands is to give you a new directory, `hw2`. You have a new main class and Makefile, and new directories `tests2`, `Type`, and `Typechecker`.

2 A Tour of Some of the Directories and Files

tests2: This directory contains an expanded set of test programs.

minijava/parser: This directory provides classes Parser and ParserException, along with some helper classes.

minijava/node: This directory contains classes for all of the different kinds of tokens and nodes that can appear in a parse tree. Node is the superclass of all of the node classes, and Token (a subclass of Node) is the superclass of all of the tokens. Classes with names beginning with "T", e.g. TPlus, are associated with particular kinds of tokens. Classes with names beginning with "P", e.g. PStmt or PExpr50, are associated with variables in the grammar. All of the "P" classes are abstract and are instantiated by "A" classes, e.g. AIfStmt or AWhileStmt. There is a distinct "A" class for each possible "right-hand side." The names for all of the classes are derived from the names of the variables and the labels, e.g. {while}, that appear in the grammar file.

(For any variable, there may be one production without a label. For example, our grammar has a single production from variable *type*, and the right-hand side of the production has no label. In this case, there is a class PType for the variable and a class AType for the right-hand side.)

The root of the parse tree is a special node of type Start. It has a single child corresponding to the variable (in our case *program*) that is the real starting symbol for the grammar.

analysis/AmhTraversal.java: This is a class that is automatically generated by our local version of SableCC. It does a traversal of parse tree. For each "A" node, for example AIfStmt, there is a method that looks like this:

```
void process(AIfStmt n) {
    n.getIf();                          // yields TIf
    n.getLparen();                      // yields TLparen
    process(n.getExpr());               // process(PExpr)
    n.getRparen();                      // yields TRparen
    process(n.getThenclause());         // process(PStmt)
    n.getElse();                        // yields TElse
    process(n.getElseclause());          // process(PStmt)

    throw new UnsupportedOperationException ();     // remove when method is complete
}
```

The various "get" methods retrieve the subparts of the right-hand side. The "process" calls on subparts lead to the traversal of the whole tree.

Let's think about the example above. The call to *n.getThenclause()* returns a PStmt. Recall, however, that PStmt is an abstract class, so the "then" child is actually something like an AWhileStmt. To distinguish among the possible kinds of statements that might appear as a child, the following method is included:

```
void process(PStmt n) {
    if (n instanceof AWhileStmt) process((AWhileStmt)n);
    else if (n instanceof ADeclStmt) process((ADeclStmt)n);
    else if (n instanceof ABlockStmt) process((ABlockStmt)n);
    else if (n instanceof AIfStmt) process((AIfStmt)n);
    else if (n instanceof AExprStmt) process((AExprStmt)n);
    else if (n instanceof AReturnStmt) process((AReturnStmt)n);
    else if (n instanceof APrintStmt) process((APrintStmt)n);
    else if (n instanceof AEmptyStmt) process((AEmptyStmt)n);
    else
        throw new RuntimeException (this.getClass() +
            ": unexpected subclass " + n.getClass() + " in process(PStmt)");

    throw new UnsupportedOperationException ();     // remove when method is complete
}
```

There is a method like this for each of the "P" classes.

In a little while, we'll talk about how you can copy `AmhTraversal.java` and use it in typechecking.

minijava/Type/Type.java: The abstract class Type should, I propose, be the superclass of all objects that we create to represent types. For this assignment, you don't have to worry about the meaning of the methods in this class. I've predefined classes for five primitive types, and created a single instance of each. For example, Type.intType refers to a Type object for ints.

minijava/Typechecker/Var.java: This is a class that you'll want to develop to hold information about a variable in the input program.

minijava/Typechecker/Method.java: This is a class that you'll want to develop to hold information about a method in the input program.

minijava/Typechecker/TypecheckerException.java: Your compiler should throw this kind of exception when it discovers a type error.

minijava/Typechecker/Typechecker.java: This is the main class for the task of typechecking. It includes some ideas that you'll probably want to use. Note the list of three data structures to keep track of types, static variables, and methods. Note also that I've preloaded the type map with four primitive types.

Eventually you'll want to uncomment the line in phase1() to create a Phase1 object, which will contain code for the first traversal of the parse tree. The remaining methods will probably need to be filled in, too. The phase 1 code can call them to accomplish the indicated tasks.

minijava/Main2.java: The new version main class is probably self-explanatory at this point. It creates a Lexer, then a Parser, and then a Typechecker. Invokes the phase1() method in the Typechecker object.

3 Your Goal

Your goal is to create records of all of the class variables and all of the methods in the input program. You should detect all of the type errors that can be detected without looking inside methods. Assuming that there are no errors, you should print information about the variables and methods that were found.

We'll talk in class about why it makes sense to construct the first phase in this way.

The various "Bad" files in the new test2 directory illustrate the kind of errors that you should be able to detect. (Bad1.java and Bad2.java contain lexical errors, not type errors.)

4 Things You'll Probably Want to Do

Copy **analysis/AmhTraversal.java** into **Typechecker/Phase1.java**. Change the class and package names in the new file. Add a constructor to match the call made in the Typechecker class. Uncomment the call to process() in method phase1() in Typechecker. Move up in the directory structure to **hw2** and then do "make" to compile everything. If there's a problem, find and fix the problem.

Create, now or later, another subclass of Type to represent arrays. Put this in a file of its own; don't make it an inner class.

Here are some ideas about how to work in **Phase1.java**:

- Leave the process(Node) method as is. This method will detect the problem if you try to run process() for some kind of Node not covered by the other cases. (Actually you probably could remove it and let the compiler detect the problem. Its inclusion here is an artifact of the development process that I used before the AmhTraversal skeleton was available.)
- Leave the process(Start) method as is. It's the first method called during the traversal.

- In process(PProgram), remove the throwing of the UnsupportedOperationException. You'll do this in every method that you're satisfied with. You can even simplify the method further by letting it contain a single line:

  ```
  process((AProgram)n);
  ```

  This is possible because there is a single right-hand side (represented by AProgram) for variable "program" (represented by PProgram).

- In process(AProgram), remove the lines that retrieve all of the tokens, with the possible exception of the program's name. Keep the "for" loop. Remove the "throw". Do you see why this all makes sense?

- In process(PMaindecl), remove the second throw but keep everything else.

- Modify a number of process methods so that they return Type, not void. process(PParam) is an example of one where this is appropriate. Modify others to return `List<Type>`, for example, process(PParamList). The point is that processing of some parts of the parse tree returns information that is needed higher up in the tree.

- You can eliminate the process() calls that take the traversal into the bodies of methods. Indeed, there are many methods that you can take out completely because they pertain only to nodes that appear within methods. (Warning: don't remove the two methods at the very bottom, which process PEmptyDim and AEmptyDim, because they are used the grammar for types.)

- Start early! I'm serious.

- Recompile frequently! Even if there are moments where you choose to defer running tests because the program is in an intermediate state, there's a lot you can learn by trying to recompile.

5 Submitting Your Work

Write a short explanation of your work on this assignment, being sure to describe any interesting choices that you made. Place this file in your hw2 directory, and then create a tar or zip file. Submit your work electronically at `www.cs.amherst.edu/submit`. There is nothing to submit on paper this time.