Programming Assignment III

A Parser for COOL

Due Monday, October 20, 2014

1. Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator (the Java tool is called `CUP`) and a package for manipulating trees. The output of your parser will be an abstract syntax tree (AST). You will construct this AST using semantic actions of the parser generator.

You certainly will need to refer to the syntactic structure of Cool, found in **Figure 1** of the **CoolAid** manual available at the course website, as well as other portions of the reference manual.

Documentation for `CUP` may be found online at:
http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html

The documentation for the tree package is described in the Cool JavaDoc available online at:
http://ece.uprm.edu/~bvelez/courses/Fall2014/icom4029/cooldoc/

(browse to the "`TreeNode`" class)

You will need the tree package information for this and future assignments.

There is a lot of information in this handout, and you need to know most of it to write a working parser. Please read the handout thoroughly.

You must work in a group for this assignment (where a group consists of two people).

2. Setting Up the Base Code

For this phase of your project you will be using the files in the **`assignments/PA3J`** directory in the Cool distribution code that you already downloaded and unpacked for previous assignments. Begin by creating a new directory named MyPA3J within the **`assignments`** directory. You will put your source files in this directory.

```
> cd ~/cool
> mkdir MyPA3J
> cd MyPA3J
```

You must now edit the **`Makefile`** inside the original **`PA3J`** directory to make it point to your cool source code directory. Replace the text that reads **`/home/you/cool`** by the pathname to your cool directory. If you extracted the code in your home directory you may simply use **`~/cool`** as the new pathname.

Next copy all the relevant **`PA3J`** files from the Cool source distribution into your **`MyPA3J`** directory by using the make command and the **`Makefile`** that you edited before:

```
> cd ~/cool/assignments/MyPA3J
> make -f ~/cool/assignments/PA3J/Makefile source
```

Your must now rename two files in your **`MyPA3J`** directory to avoid conflicts with Eclipse:

```
> cd ~/cool/assignments/MyPA3J
> mv README README-PA3J
> mv cool-tree.java CoolTree.java
```

Open Eclipse on the same workspace used for PA2. **Make sure that you have tagged the CoolCompiler project with the version of the lexer that you submitted for grading before you do any of the following modifications to the project. Otherwise you may no easily recover the status of the lexer that was ready for grading.** Only one of the project members needs to tag the project. We recommend that you visit GitHub with a browser and verify that the tag exists in the remote repository.

Remember to tag your lexer code in order to be able to go back in time to a working version of your lexer if needed (see PA2 instructions). Proceed by copying all the Java (.java extension) files into the default package on your Eclipse **`CoolCompiler`** project. There is no need to create a new project. Some of these files may already be part of your project as they were used during the lexer phase. It is ok to replace them since you should not have modified them for PA2.

There are a few additional files that you should copy to the root of the **`CoolCompiler`** project: **`cool.cup`**, **`good.cl`**, **`bad.cl`**, and **`README-PA3J`**. As soon as you copy **`cool.cup`** the JFlex plugin will automatically generate a new **`CoolLexer.java`** file in your default package. This file contains the class that does the actual lexical analysis. Confirm that this is the case. You should never modify this file directly as JFlex refreshes it every time you modify and save **`cool.cup`**.

The next step requires you to refactor/rename three classes in the **`CoolTree.java`** file to avoid conflicts with Eclipse. To refactor/rename a class just right-click on the class name that appears on its class declaration and select **Refactor -> Rename**. The name of the class will be highlighted allowing you to replace the name as follows:

- Rename **`Class_`** to **`ClassAbstract`**
- Rename **`Program`** to **`ProgramAbstract`**
- Rename **`Formal`** to **`FormalAbstract`**

If you attempt to rename the classes using Find/Replace you may have a hard time finding all references to these classes across multiple files. Refactor/rename does this automatically. You should learn to use your software development tools wisely to achieve maximal productivity.

3. Files and Directories

Your **`MyPAJ3`** directory now contains a number of files some of which were copied read-only (using symbolic links). **You should not edit these files**. In fact, if you make and modify private copies of these files, you may find it impossible to complete the assignment. See the instructions in the **`README`** file. The files that you will need to modify are:

- **`cool.cup`**

This file contains a start towards a parser description for Cool. You will need to add more rules. The declaration section is mostly complete; all you need to do is add type declarations for new nonterminals. (We have given you names and type declarations for the terminals.) The rule section is very incomplete.

- `good.cl` and `bad.cl`

  These files test a few features of the grammar. You should add tests to ensure that `good.cl` exercises every legal construction of the grammar and that `bad.cl` exercises as many types of parsing errors as possible in a single file. Explain your tests in these files and put any overall comments in the `README` file.

- `README`

  As usual, this file will contain the write-up for your assignment. Explain your design decisions, your test cases, and why you believe your program is correct and robust. It is part of the assignment to explain things in text, as well as to comment your code.

Follow the instructions at the end of this document to turn in your assignment, not the ones in the `README` file.

4. Testing the Parser

The main class of your parser phase is `Parser.java`. In order to make this class use your own lexer, you must rename all references to `CoolTokenLexer` within its `main` method to references to `CoolLexer`, the name of the lexer class generated by the JFlex plugin from your `cool.lex` specification developed in PA2. To run your parser you can run `Parser.java` as a Java application and paste Cool code into the console to test it. To use the reference lexer use the `CoolTokenLexer` class in place of the `CoolLexer` class in `Parser.java` and paste lexer output into the console to test your app. You can generate sample lexer output by running the `reference-lexer` available on the `cool/bin` directory with some Cool input file.

To complete your parser you must add grammar rules to the CUP specification in the `cool.cup` file with appropriate LALR reduce rules that would build the Abstract Syntax Tree (AST) from the bottom up as studied in class. The top call to `parser.parse()in Parser.java` returns the root of the AST representing the entire input cool program. `Parser.java` will then dump a textual representation of the tree to the console which you should make sure is as similar as possible to the output generated by the reference parser (`ref-parser`) available in your PA3J directory.

You may run your Parser.java app adding a `-p` flag to your run configuration for debugging the parser. Using this flag causes lots of information about what the parser is doing to be printed on stdout. CUP produces a human-readable dump of the LALR(1) parsing tables in the `cool.output` file. Examining this dump is frequently useful for debugging the parser definition.

You should test this parser on both good and bad inputs to see if everything is working. Remember, bugs in your parser may manifest themselves anywhere.

Your parser will be graded using the **`reference-lexer`**. Thus, even if you do most of the work using your own lexer you should configure and test your parser with the reference-lexer before tagging your project for grading.

5. Parser Output

Your semantic actions should build an AST. The root (and only the root) of the AST should be of type **`program`**. For programs that parse successfully, the output of **`Parser.java`** is a listing of the AST.

For programs that have errors, the output is the error messages of the parser. We have supplied you with an error reporting routine that prints error messages in a standard format; please do not modify it. You should not invoke this routing directly in the semantic actions; **`CUP`** automatically invokes it when a problem is detected.

Your parser need only work for programs contained in a single file. You don't have to worry about compiling multiple files.

6. Error Handling

You should use the **`error`** pseudo-nonterminal to add error handling capabilities in the parser. The purpose of error is to permit the parser to continue after some anticipated error. It is not a panacea and the parser may become completely confused. See the **`CUP`** documentation for how best to use error. In your **`README`**, describe which errors you attempt to catch. Your test file **`bad.cl`** should have some instances that illustrate the errors from which your parser can recover. To receive full credit, your parser should recover in at least the following situations:

- If there is an error in a class definition but the class is terminated properly and the next class is syntactically correct, the parser should be able to restart at the next class definition.
- Similarly, the parser should recover from errors in features (going on to the next feature), a let binding (going on to the next variable), and an expression inside a {...} block.

Do not be overly concerned about the line numbers that appear in the error messages your parser generates. If your parser is working correctly, the line number will generally be the line where the error occurred. For erroneous constructs broken across multiple lines, the line number will probably be the last line of the construct.

7. Remarks

You may use precedence declarations, but only for expressions. Do not use precedence declarations blindly (i.e. do not respond to a shift-reduce conflict in your grammar by adding precedence rules until it goes away). If you find yourself making up rules for many things other than operators in expressions and for let, you are probably doing something wrong.

The Cool `let` construct introduces an ambiguity into the language (try to construct an example if you are not convinced). The manual resolves the ambiguity by saying that a `let` expression extends as far to the right as possible. The ambiguity will show up in your parser as a shift-reduce conflict involving the productions for `let`.

This problem has a simple, but slightly obscure, solution. We will not tell you exactly how to solve it, but we will give you a strong hint. In `coolc`, we implemented the resolution of the `let` shift-reduce conflict by using a `CUP` feature that allows precedence to be associated with productions (not just operators). See the `CUP` documentation for information on how to use this feature.

Since the `Cool` compiler may use pipes to communicate from one stage to the next, any extraneous characters produced by the parser can cause errors; in particular, the semantic analyzer may not be able to parse the AST your parser produces.

8. Notes

You must declare **`CUP`** “types" for your non-terminals and terminals that have attributes. For example, in the skeleton **`cool.cup`** is the declaration:

```
nonterminal program program;
```

This declaration says that the non-terminal **`program`** has type **`program`**.

It is critical that you declare the correct types for the attributes of grammar symbols; failure to do so virtually guarantees that your parser won't work. You do not need to declare types for symbols of your grammar that do not have attributes.

The **`javac`** type checker (used by Eclipse) complains if you use the tree constructors with the wrong type parameters. If you fix the errors with frivolous casts, your program may throw an exception when the constructor notices that it is being used incorrectly. Moreover, **`CUP`** may complain if you make type errors.

8. Turning In the Assignment

1. Make sure your code is in **`cool.cup`** and that it compiles and works.
2. Your test cases should be in **`good.cl`** and **`bad.cl`**. Their output should be in **`good.output`** and **`bad.output`**, respectively (these are generated by executing **`gmake`**).
3. Include any other relevant comments in the **`README`** file and answer any questions that appear in it.
4. Make sure everything (**`cool.cup`**, **`good.cl`**, **`bad.cl`**, **`good.output`**, **`bad.output`**, and **`README`**) is updated in your remote GitHub repository.
5. Use Eclipse to tag your project at the stage that you want to submit for grading and send your tag to Bienvenido.Velez@upr.edu so we can pull your parser for grading. Your tag should reflect a date on or before the project deadline (October 20, 2014).

HAVE FUN AND LEARN A LOT!!