CS 331 Compilers **Fall 2017**

Programming Project 2: Parser

Prof. Szajda *Due Thursday, October 12, 11:59:59 pm*

NOTE: I **highly** recommend you begin this when you get it. There are terms in here (e.g., shift-reduce conflict) that we have not yet covered. We will soon. But as with the prior project, there is much documentation to be read. You should start on that now!

1 Introduction

In this project you will write a parser for Cool. The project makes use of two tools: the parser generator (the Java version of the 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 Cool Reference Manual (as well as other parts). The documentation for `CUP` is available online. The documentation for the Java version of the tree package is available online via a link on the course web page. You will need the tree package information for this and future projects.

There is a lot of information in this handout, and you need to know most of it to write a working parser. So, as with all the programming projects, *you must read the handout thoroughly.* Also as before, you must work in pairs for this project.

2 Files and Directories

To get started, create a directory (in the virtual machine) where you want to do the project and execute the following command in that directory

```
make -f /usr/class/cs143/assignments/PA3J/Makefile
```

(and as with last project, notice the J in the path name, and that the project number is one greater than the project number for our class). This command will copy a number of files to your directory. Some of the files will be 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 project. 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. The declaration section is mostly complete, but you will need to add additional type declarations for new nonterminals you introduce. I have given you names and type declarations for the terminals. You might also need to add precedence declarations. The rule section, however, is rather incomplete. I have provided some parts of some rules. You should not need to modify this code to get a working solution, but you are welcome to if you like. However, do not assume that any particular rule is complete.

- `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 project. Explain your design decisions, your test cases, and why you believe your program is correct and robust. It is part of the project to explain things in text, as well as to comment your code.

3 Testing the Parser

You will need a working scanner to test the parser. You may use either your own scanner or the `coolc` scanner. By default, the `coolc` scanner is used; to change this behavior, replace the `lexer` executable (which is a symbolic link in your project directory) with your own scanner. Don't automatically assume that the scanner (whichever one you use!) is bug free – latent bugs in the scanner may cause mysterious problems in the parser.

You will run your parser using `myparser`, a shell script that "glues" together the parser with the scanner. Note that `myparser` takes a `-p` flag for debugging the parser; using this flag causes lots of information about what the parser is doing to be printed to `stdout`. `CUP` produce a human-readable dump of the LALR(1) parsing tables in the `cool.output` file. Examining this dump may sometimes be useful for debugging the parser definition.

You should test this compiler 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 my lexical analyzer. Thus, even if you do most of the work using your own scanner you should test your parser with the `coolc` scanner before turning in the assignment.

4 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` is a listing of the AST.

For programs that have errors, the output is the error messages of the parser. I 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 routine directly in the semantic actions; `CUP` automatically invokes it when a problem is detected.

For constructs that span multiple lines, you are free to set the line number to any line that is part of the construct. Do not worry if the lines reported by your parser do not exactly match the reference compiler. Also, your parser need only work for programs contained in a single file – dont worry about compiling multiple files.

5 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.

6 The Tree Package

The documentation for the Java version of the tree package for Cool abstract syntax trees is available on the course web page (link is `http://web.stanford.edu/class/archive/cs/cs143/cs143.1112/javadoc/cool_ast/`). Start with the documentation for class `TreeNode`. You will need most of that information to write a working parser.

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).

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. Depending on the way your grammar is written, this ambiguity *may* show up in your parser as a shift-reduce conflict involving the productions for `let`. If you run into such a conflict, you might want to consider solving the problem 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 `mycoolc` compiler uses 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. **Since any prints left in your code will cause you to lose many points, please make sure to remove all prints from your code before submitting the assignment.**

8 Java Related 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 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.

9 Submission

To submit your project, use send an email containing your single submission file as an attachment, to the address I have supplied (see the Blackboard file with these email addresses). Your submission file should contain your `cool.cup` file, your `README` file, and all of the `.cl` files you used to test your parser.