CS164 Programming Language and Compilers **Spring 2016**

Programming Assignment III

Assigned: February 8 **Due:** March 3, 2016 at 11:59 PM

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, on page 16, of the Cool manual (as well as other parts). There is a documents on CUP online. There is also a section (Dragon Book 4.9) in the textbook on yacc, a close predecessor of CUP. The tree package is described in the online documentation. 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 one or two people). The submit program will ask you to specify group members when you turn in your assignment.

2 Files and Directories

Everything you need is included in the archive file (PA3.zip) available from the course website. This package contains a number of files. Some of the files (Java files) should not be edited. In fact, if you modify these files, you may find it impossible to complete the assignment. See the instructions in the README file.

2.1 Files to modify

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. We 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. We 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 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.

2.2 Helper scripts

The assignment package includes several scripts to easily compile and run your lexer (and the reference implementation as well). The following is the list of commands you can run:

- To compile your parser implementation: `ant parser`
- To run your parser on the file good.cl and bad.cl: `ant test`
- To run ten examples in tests directory: `ant test-all`
- To run your parser on an input file 'foo.cl': `python myparser.py foo.cl`
- To compile a program 'foo.cl' with your parser and the rest of the reference compiler: `python mycoolc.py foo.cl`
- To run the reference parser on an input file 'foo.cl' : `python parser.py foo.cl`
- To compile a program 'foo.cl' with the reference compiler and to generate 'foo.s': `python coolc.py foo.cl`
- To run the compiled program 'foo.s' using the MARS mips simulator: `python runmips.py foo.s`

3 Testing the Parser

You will run your parser using myparser.py, a script that "glues" together the parser with the reference scanner (i.e., lexer). Note that myparser.py takes a -p flag for debugging the parser; using this flag causes lots of information about what the parser is doing to be printed on `stdout`. CUP produce 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.

Once you are confident that your parser is working, try running mycoolc.py to invoke your parser together with other compiler phases. 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.

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. 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 routine 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—don't 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 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 is available on the course web page (*Cool support files JavaDoc*). You will need most of that information to write a working parser.

7 Notes for Implementation

- 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. 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 mycoolc.py 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.

- 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.

8 What to Turn In

When you are ready to turn in the assignment, first upload your assignment to the instructional account, then type ant submit-clean in the directory where you have uploaded your assignment. This action will remove all the unnecessary files, such as object files, class files, Emacs autosave files, etc. Following ant submit-clean, type submit PA3 which will ask you for the names of the partners doing the assignment and will then send README, cool.cup, good.cl, bad.cl, good.output and bad.output (the outputs of running your program on good.cl and bad.cl) to the reader. The submit program will also ask you if you want to turn in any other files in the project directory. Your default answer should be "no", unless you really want us to see those files.

Doctoring the output that is sent is considered cheating (and not effective, since we test your program ourselves). If you want to explain something, do it in the README file.

The last submission you do will be the one graded. Each submission overwrites the previous one. Remember that there is a 1% penalty per hour for late assignments. The burden of convincing us that you understand the material is on you. Obtuse code, output, and write-ups will have a negative effect on your grade. Take the extra time to clearly (and concisely!) explain your results.

9 Grading (out of 50)

The point breakdown for PA3 is as follows:

- 38 points - for autograder tests
- 4 points for the README

  - 4 - thorough discussion of design decisions (e.g., the handling of let) and choice of test cases; a few paragraphs of coherent English sentences should be fine
  - 2 - vague or hard to understand; omits important details
  - 0 - little to no effort

- 4 points - for good.cl and bad.cl
  - 4 - wide range of test cases added, stressing most Cool features and most of the error conditions discussed in the handout
  - 2 - added some tests, but the scope not sufficiently broad
  - 0 - little to no effort

- 4 points - for code cleanliness
  - 4 - code is mostly clean and well-commented
  - 2 - code is sloppy and/or poorly commented in places
  - 0 - little to no effort to organize and document code