Programming Assignment III
A Parser for COOL
Due Monday, October 11, 2004

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://amadeus.ece.uprm.edu/~rbadia/cupmanual

The documentation for the tree package is described in the Cool JavaDoc available online at:

> http://amadeus.ece.uprm.edu/~rbadia/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. Files and Directories

To get started, create a directory where you want to do the assignment and execute the following command in that directory:

> **gmake -f ~icom4029/cool/assignments/PA3J/Makefile source**

(notice the "J" in the path name). 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 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.

Important: All software supplied with this assignment is supported on the Linux/Pentium platform available at the Amadeus computer center.

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

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 that, 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 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 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 our 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. 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.

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. 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 `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.

7. 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.

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 in a directory called `PA3`.

5. Create a tar-gzipped file `PA3.tar.gz` containing the `PA3` directory:

    `tar -czf PA3.tar.gz PA3`

6. Submit the file (before October 11 11:59pm):

    `~icom4029/submit/submit 3 PA3.tar.gz`

You must use any of the group members' class accounts when submitting the assignment. You can submit multiple times; if you do so, any previous submissions will be overwritten (until after the project's deadline).