Programming Assignment III

First Due Date: (Grammar) March 12, 2005 (submission dated midnight).
Second Due Date: (Complete) March 19, 2005 (submission dated midnight).

Purpose: This project is intended to give you experience in using a parser generator, namely bison or JavaCup, and to bring together the various issues of syntax specification, parsing, and abstract syntax tree construction discussed in class.

1 Introduction

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator (the C++ tool is called bison; 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, as well as other portions of the reference manual. There is a section on bison in the course reader. There is also a section (Dragon Book 4.9) in the textbook on yacc, a close predecessor of bison. Documentation for CUP may be found online. The C++ version of the tree package is described in the *Tour of Cool Support Code* section in the back of the Cool manual, while the documentation for the Java version is also available online. 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 may 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 one of the following commands *in that directory.* For the C++ version of the assignment, you should type

```
gmake -f ~pollock/public/cool02/assignments/PA3/Makefile
```

For Java, type:

```
gmake -f ~pollock/public/cool02/assignments/PA3J/Makefile
```

(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.y (in the C++ version) / cool.cup (in the Java version)
  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. You are welcome to create a set of test cases good0.cl, good1.cl,... and bad0.cl, bad1.cl,... if you want instead of single good and bad cases.

- 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 both Solaris SPARC and Solaris x86. Remember to run gmake clean if you switch architectures.

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`. Both bison and 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 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.

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; `bison`/`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 bison/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 $\{\ldots\}$ 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.

NOTE: The more error nonterminals you add to the grammar, the more likely you will create grammar conflicts that did not occur before, so I suggest that you start with only two error nonterminals in your grammar, and add more if you want to give better error handling. When you start creating conflicts, stop adding error nonterminals.

6 The Tree Package

There is an extensive discussion of the C++ version of the tree package for Cool abstract syntax trees in the *Tour* section of the Cool manual. The documentation for the Java version is available on the course web page. 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). 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 `bison/CUP` feature that allows precedence to be associated with productions (not just operators). See the `bison/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.

8 Notes for the C++ version of the assignment

If you are working on the Java version, skip to the following section.

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

  ```
  %type <program> program
  ```

  This declaration says that the non-terminal `program` has type `<program>`. The use of the word "type" is misleading here; what it really means is that the attribute for the non-terminal `program` is stored in the `program` member of the `union` declaration in cool.y, which has type `Program`. By specifying the type

  ```
  %type <member_name> X Y Z ...
  ```

  you instruct **bison** that the attributes of non-terminals (or terminals) `X`, `Y`, and `Z` have a type appropriate for the member `member_name` of the union.

  All the union members and their types have similar names by design. It is a coincidence in the example above that the non-terminal `program` has the same name as a union member.

  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 g++ type checker complains if you use the tree constructors with the wrong type parameters. If you ignore the warnings, your program may crash when the constructor notices that it is being used incorrectly. Moreover, bison may complain if you make type errors. Heed any warnings. Don't be surprised if your program crashes when bison or g++ give warning messages.

9 Notes for the Java version of the assignment

If you are working on the C++ version, skip this section.

- 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.

What to Turn in:

For deadline 1: submit via course submission website

For deadline 2: submit via course submission website

Deadline 1: For the first deadline, you should have a complete bison specification for the Cool language, but need not have trees being constructed. Your parser should accept correct Cool programs, and print error messages for syntactically erroneous Cool programs. Your good.cl and bad.cl should be included in this deadline submission. The README for this submission need only include some text on your overall approach to the grammar construction and dealing with conflicts. You do not need to do any error recovery for this first deadline.

Note that until you create the actions to create the ast, the test of your parser through the makefile command will not run properly as it is expecting an ast to be produced. As long as you can see that it has parsed the whole program and reduced to the start symbol, your grammar is working ok for this run of test.cl.

Deadline 2: For the second deadline, your parser should also perform AST construction in order to satisfy the full specification of this assignment sheet. Error recovery should be included in the submission for this deadline. You are welcome to change your good.cl and bad.cl for this submission. The README documentation should be more extensive and complete for this submission.

Criteria for Grading:

(28) Grammar:

- accepts syntactically correct programs without error messages/bombing Which features of the languages are NOT accepted properly by parser?
- does not accept syntactically erroneous programs Which features are improperly accepted when they should be flagged as errors?
- precedence and associativity of operators consistent with language definition

(5 pts) Error Handling:

- detection and descriptive error messages printed out: error explanation and maybe location (no error recovery for first deadline is needed, just detection of single errors) For programs that have errors, the output is the error messages of the parser.

(5 pts) Test cases:

- good.cl extended with pretty thorough testing of correct program structure - bad.cl extended to test parsing errors If want to add other test cases, that is fine also.

(2 pts) External documentation in README. Explain your design decisions, your test cases, and why you believe your program is correct and robust.