

Phase II: The Parser

First Due Date: (Grammar) October 14th.

Second Due Date: (Complete) October 21st.

Teamwork: Highly encouraged.

1 Overview

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

In this assignment you will write a parser for Cool. The assignment makes use of two tools: the parser generator (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 manual, as well as other portions of the reference manual. 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.*

2 Teamwork

You should try to find a partner to do this project with. This will improve your team-skills and (hopefully) you will get done faster. As soon as you found a partner to work with, both of you should announce this decision to the TA. The TA will then enable both of you to see either of your folders. Choose one of your folders to work with. Svn is a good way to share source files in a team and to keep each other updated on the changes made. Furthermore, many svn-clients provide tools to resolve conflicts of changes made to a particular file efficiently.

Teams of three people can be established after prior approval of the professor.

Teams are established for each homework problem separately. If you want to continue working in your team (which is strongly encouraged), please announce this to the TA after submission of this homework. In the case that your team breaks up before finishing the assignment, each of you will be required to submit a completed assignment (in this case, your code is obviously allowed to be very similar). Extensions of the deadline(s) may be given if you provide good reasons for breaking up with your partner.

3 Tasks

1. Read the Java CUP specification manual at <http://www2.cs.tum.edu/projects/cup/manual.html>
2. The file `parser/cool.cup` 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.
3. Make yourself familiar with the different classes derived from `TreeNode` (they are all in `treeNodes` package). Objects of these classes will be the nodes of the AST that you are asked to create.

4. As usual, the file `README_Phase2.txt` contains detailed instructions for the assignment. You should also edit this file to include the write-up for your project. Explain your design decisions 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.

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. 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 Notes

- 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.

- 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.

7 Submission

When you are ready to turn in the assignment, commit your files to the svn-repository. You may also commit several versions. If you do not want the version of the exact due date to be used for grading, please email the TA about what version should be used instead. You may submit more than one test case if you like.

Deadline 1: For the first deadline, you should have a complete CUP 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. 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.

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. The README documentation should be more extensive and complete for this submission.

8 Evaluation Criteria

Correct Programs:

- accepts syntactically correct programs without error messages/bombing 20%
- generates correct AST 30%
- precedence and associativity of operators consistent with language definition 15%

Incorrect Programs:

- does not accept syntactically erroneous programs 15%
- descriptive error messages and location of error 5%

Documentation 5%

Clarity 10%