Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2000 Segment II: Syntactic Analysis – Project Assignment Monday, September 11

DUE: Tuesday, September 26

Your parser must be able to correctly parse programs that conform to the grammar of the Decaf/Espresso language[1]. Any program that does not conform to the language grammar must be flagged with at least one error message.

This part of the project involves the following tasks:

1. Write a suitable grammar for the Decaf/Espresso language. You will need to transform the reference grammar (along with the precedence rules) in Handout 6 or Handout 7 into a grammar expressed in the BNF-like input syntax of CUP, the LALR(1) parser generator that you will be using for the project. Documentation for CUP is linked to from the 6.035 web page.

   A suitable grammar is an LALR(1) grammar that CUP can process with no conflicts. The actions should print out the productions being applied during the parse if the debug flag is turned on. We have modified CUP to provide a String, `PRODSTRING`, in the CUP actions which represents the current reduction and can be used in generating the debugging output.

2. Have a driver that parses the command line (using `java6035.tools.CLI.CLI`) and runs your parser if appropriate (or your scanner, if that is what is requested). The debug flag should apply only to the last stage requested, so using it for the parser shouldn't cause the scanner to list the tokens. The driver should be called Compiler.class in the root of your class heirarchy, so you can run it, given the right CLASSPATH, with

   ```
   java Compiler <infile>
   ```

   We encourage you to use the provided CLI tool instead of writing your own command line parser. Documentation for CLI is linked to from the 6.035 homepage.

3. Your parser should also include basic syntactic error recovery, by adding extra error productions that use the error symbol in CUP. Do not attempt any context-sensitive error recovery. Keep it simple. We do not require the error recovery to go beyond the ability to recover from a missing right brace or a missing semicolon.

You should select one group member's scanner to use for the rest of the term. If you do not like your scanner, you can use a scanner that we will provide. Your parser does not have to, and should not, recognize context-sensitive errors *e.g.,* using an integer identifier where an array identifier is expected. Such errors would be detected by the static semantic checker. Do not try to detect context-sensitive errors in your parser!

[1] If, for some reason, you implement any language extensions, please ensure that the extensions are supported under some option and not by default. You will not receive any grade credit for implementing language extensions.

You may obtain additional information on CUP by browsing the CUP links on the 6.035 home page or by reading Section 3.4 of the "Tiger" book, and by attending the recitation. In addition, you might want to look at Section 3 of the "Tiger" book, or Sections 4.3 and 4.8 in the Dragon book for tips on getting rid of shift/reduce and reduce/reduce conflicts from your grammar.

To help get you started, we have provided you with a template for the cup file in

```
/mit/6.035/provided/parser/template.cup
```

which you should copy to your team directory, rename to {decaf/espresso}.cup and fill in with the appropriate productions and actions. In order to run CUP on this file, execute the following commands:

```
setenv CLASSPATH /mit/6.035/classes:$CLASSPATH
java java_cup.Main < {decaf/espresso}.cup
```

Only needs to be run once per shell

What to Hand In

Follow the directions given in Handout 4 when writing up the hard copy for your project. Include a full description of how your parser is organized. In addition, please provide a listing of your grammar (the input to the parser generator). Do not resubmit your scanner code in your hard copy; if you made any changes to your scanner to make it compatible with your parser, include a brief note that describes them.

The electronic copy portion of the hand-in procedure is similar to that of the Scanner segment. Provide a gzipped tar file named `leNN-parser.tar.gz` in your group locker, where `NN` is your group number.. This file should contain all relevant source code and a `Makefile`. Additionally, you should provide a Java archive, produced with the `jar` tool, named `leNN-parser.jar` in the same directory. (All future hand-ins will be roughly identical, except with different names for the files.)

Unpacking the tar file and running `make` should produce the same Java archive. With the `CLASSPATH` set to `leNN-parser.jar:/mit/6.035/classes`, you should be able to run your compiler from the command line with:

`java Compiler` <*filename*>

Your compiler should handle both `scan` and `parse` as arguments to the `-target` flag, and it should assume `parse` by default.

The resulting output to the terminal should be a report of any syntax errors encountered while parsing the file. Error messages should include the line number of the error. Any syntactically incorrect program should be flagged with at least one error message. Multiple error messages should be output for programs with multiple syntax errors that are amenable to error recovery (e.g., a missing right brace or a missing semicolon).

In addition, your parser should have a debug mode in which the productions being applied during the parse are printed in some form. This can be run from the command line by,

`java Compiler -debug` <*filename*>

Some sample code has been provided in the `6.035` locker to help you implement this interface. `/mit/6.035/provided/parser/Makefile` can be copied to your working directory. It will create the correct tar and jar files.

A test script has also been provided to check that the tar and jar files exist, and runs the parser in the jar file using the standard interface. It does *not* actually check the output of your compiler for correctness. It just tries to run it and tests that the tar and jar files are in their proper locations. This script can be run by typing `add 6.035; run_parser` <*filename*>.

Test Cases

The test cases provided for the parser are in `/mit/6.035/tests/parser/{decaf/espresso}/`

If your parser passes all of the test cases (accepts the syntactically correct programs and rejects the incorrect ones) you get 10 points of the possible 30 points for implementation. The other 20 points will be based on how well your parser performs on hidden test cases.