Decaf PP2: Syntax Analysis

Date Due: **03/17/2018 Sunday 11:59pm**

1 Goal

In this programming project, you will extend the Decaf compiler to handle the syntax analysis phase, the second task of the front-end. The parser will read Decaf source programs and construct an Abstract Syntax Tree (AST). If no syntax errors are found, your compiler will print the completed AST at the end of parsing. At this stage, you are not responsible for verifying semantic rules, just the syntactic structure.

2 Syntactical Structure of Decaf

The reference grammar given in the Decaf language handout defines the official grammar specification you must parse. The language supports global variables and functions, constructs such as if, while, etc. First, read the grammar specification carefully. Although the grammar is fairly large, most of is not tricky to parse.

3 Testing

In the starting project, there is a *samples* directory containing various input files and matching *.out* files which represent the expected output that you should match. $diff$ is your friend here!

Note that, *the provided test files do not test every possible case!* Examine the test files and think about what cases aren't covered. Make up lots of test cases of your own. Run your parser on various incorrect files to make sure it finds the errors. What formations look like valid programs but are not? What sequences might confuse your processing of expressions or class definitions? How well does your error recovery strategy stand up to abuse?

Remember that syntax analysis is only responsible for verifying that the sequence of tokens form a valid sentence given the definition of the Decaf grammar. Given that our grammar is somewhat "loose", some apparently nonsensical constructions will parse correctly and, of course, we are not yet doing any of the work for verify semantic validity (type-checking, declare before use, etc.). The following program is valid according to the grammar, but is obviously not semantically valid. It should parse correctly.

```
string binky()
{
      string b;

if (1.5 * "Stanford")
b / 4;
}
```

4 Hints

Some hints to help you code:

1. Our reference systems are the fox servers (fox01.cs.utsa.edu to fox06.cs.utsa.edu). Your assignments will be graded on these servers.

2. Google is always your best friend. You can find answers to almost all of your questions online.

5 Grading

This project is worth 10 points and points will be allocated for correctness. We will run your program through the given test files from the samples directory as well as other tests of our own, using $diff - w$ to compare your output to that of our solution.

6 Deliverables

Electronically submit your entire project to Blackboard. You should submit a tar.gz of the project directory. Be sure to include a brief README file.

Because we grade the submissions using scripts, it is important that everyone uses the same directory structure. The uploaded folder should contain your source code, all dependencies, and a sub-folder called 'workdir' where you should put two files 'build.sh', and 'exec.sh'. Running 'build.sh' should build your project, and running 'exec.sh <filepath>' should execute your compiler on a source code file at <filepath>', and all output should be written to the standard output stream, so that the grader can use 'exec.sh <filepath> > <outputpath>' to redirect your output into files and compare with the ground truth.