**Project #2 - Code Parser with Abstract Syntax Tree (AST)**

Version 1.3 (added clarifying footnote about functions and revised due date)  
Revised: 02/28/2016 12:06:34  
Due Date: Thursday March 10th

**Purpose:**

The first two projects this Spring focus on building software tools for code analysis. We will emphasize C++ code but want our tools to be easily extendable to other similar languages like C# and Java.

Code analysis consists of extracting lexical content from source code files, analyzing the code's syntax from its lexical content, and building an Abstract Syntax Tree (AST) that holds the results of our analysis. It is then fairly easy to build several backends that can do further analyses on the AST to construct code metrics, search for particular constructs, or some other interesting features of the code.

You will find it useful to look at the [Parsing](http://www.ecs.syr.edu/faculty/fawcett/handouts/Webpages/BlogParser.htm) blog for a brief introduction to parsing and code analysis.

In this second project we will build a Parser-based code analyzer in C++ that consists of at least these packages:

* **Tokenizer - from Project #1:**

extracts words, called tokens, from a stream of characters. Token boundaries are white-space characters, transitions between alphanumeric and punctuator characters, and comment and string boundaries. Certain classes of punctuator characters belong to single character or two character tokens so they require special rules for extraction.

* **SemiExpression - from Project #1:**

groups tokens into sets, each of which contain all the information needed to analyze some grammatical construct without containing extra tokens that have to be saved for subsequent analyses. SemiExpressions are determined by special terminating characters: semicolon, open brace, closed brace, newline when preceeded on the same line with '#', and colon when preceded by one of the three tokens "public", "protected", or "private".

* **Parser:**

Uses Rules to identify specific syntactical constructs and builds an Abstract Syntax Tree to hold that information during analyses. The Parser is a class that contains instances of Rule classes and passes token collections it acquires from ITokenCollection to each rule in turn until there are no more token collections to process. Rules are containers for Actions that are classes that contain methods doAction(const SemiExp& se) that use se to add new elements to the AST.

* **RulesAndActions:**

A package that contains definitions of the Rules and Actions used by Parser. This package will be modified for each type of code analysis implemented by an application.

* **AbstractSyntaxTree:**

A package that provides functionality for the actions to build an Abstract Syntax Tree. This package provides an interface for building and for extracting scope information from the tree.

* **Metric Analysis and Metric Executive:**

Application specific packages that uses information stored in the AST to create a display of all the function and method sizes and complexities of each source code package identified for analysis.

* **FileMgr:**

A package that navigates through a directory tree rooted at some specified path and returns names of all the files and/or directories matching a pattern.

In this project we will develop and test a C++ Metric Analysis program:

**Requirements:**

Your Metric Analyzer:

1. **Shall** use Visual Studio 2015 and its C++ Windows Console Projects, as provided in the ECS computer labs.
2. **Shall** use the C++ standard library's streams for all I/O and new and delete for all heap-based memory management1.
3. (**2**) **Shall** provide C++ packages for analyzing function size and complexity metrics for a set of specified packages. These packages will use the Tokenizer and SemiExpression packages you developed2 in Project #1.
4. (**3**) **Shall** provide a Parser package with a Parser class that is a container for Rules and that provides the interfaces IRule and IAction for rules contained in the Parser and actions contained in each rule.
5. (**3**) **Shall** provide an associated RulesAndActions package that has rules to detect:
   * global functions and static and non-static member function definitions3.
   * beginning and end of all C++ scopes.

and actions for each rule that support building the Abstract Syntax Tree (AST) that represents a single file's scope structure and identifies the type and line number extend of each scope.

1. (**4**) **Shall** provide a facility for building an abstract syntax tree that provides an interface for adding scope nodes to the tree and an methods to analyze the contents of the tree.
2. (**3**) **Shall** provide a FileMgr package that supports finding files and/or directories in a directory tree rooted at a specified path.
3. (**4**) **Shall** provide a MetricsAnalysis package for evaluating and displaying the size and complexity of all global functions, static member functions, and non-static member functions in each of a set of specified packages.
4. (**3**) **Shall** provide a MetricsExecutive package that enables collecting metrics for all the packages with names that match a specified pattern in a directory tree rooted at a specified path. Please define the path and file patterns on the MetricsExecutive command line.
5. (**3**) **Shall** include an automated unit test suite that exercises all of the packages provided in your submission and demonstrates that you met all requirements4.
6. That means that you are not allowed to use any of the C language I/0, e.g., printf, scanf, etc, nor the C memory management, e.g., calloc, malloc, or free.
7. You may use the instructor's solution for the scanner. Students who use their own scanner code will receive a 5 point bonus.
8. You just have to detect ALL functions. You don't have to keep track of which are global, which are static, and which are non-static.
9. This is in addition to the construction tests you include as part of every package you submit.