ENSC 251 – Summer 2017 – Course Project Part 4

Due: **Friday July 21, 2017, 6:00 pm**

For our course project so far, we have developed code for tokenization of source code conforming to a subset of the c++ language and classification of those tokens into respective classes. We also developed a generic TreeNode class for building trees for different applications.

The next phase for our course project is to create an Abstract Syntax Tree (AST) with each node representing a token. The project template has a solution for Part 2 and the TreeNode Class which was created during Part 3. You will be using a vector of tokens from token classification (Part 2) to generate an AST based on the set of grammar rules given to you in Backus–Naur Form (BNF).

The provided Eclipse project template contains the text file "**c++SubsetBNF.txt**" listing the grammar to be considered when creating the AST for an input set of classified tokens. The grammar is for a simplified version of the C++ language. The types of expressions and operators supported in this project are limited. For example, we have excluded logical and relational operators. Other elements of C++ language like functions and IF and WHILE statements are also omitted. The reason for these limitations and omissions is to simplify the course project to make things easier for you.

You can read more about BNF from:
https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form
http://www.cs.utsa.edu/~wagner/CS3723/grammar/examples2.html

In order to parse the tokens and generate an AST from them, we will be using recursive descent parsing based on the grammar rules specified in "**c++SubsetBNF.txt**". Recursive descent parsing uses a set of functions that are written based upon the BNF grammar. These functions return sub-trees in "bottom to top" fashion so that you eventually get the entire AST as a return value for the initial function used for parsing.

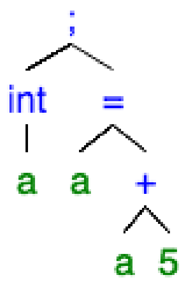
Once you have generated an AST, we want you to traverse that tree based on the traversal functions developed in part 3 of the project. Once again, you will be utilizing the pair of function pointers – one which traverses the tree and the other which will print the content of the node. For a certain class of token, you may need to specify what type of traversal should be used for that node. The goal of this traversal is to regenerate C++ language code that is equivalent from a language perspective to the original source stream. The "spacing" of tokens may be different, but both the original source stream and the regenerated code should be able to be parsed into identical ASTs. You need to make sure that the output of traversal contains all the tokens even if some, like ":", are not stored in the AST.

NOTE: undeclared variables (e.g, errno) are all classified as **int_id**.

Example 1: if your input source code is:

int a; a=a+5;

The generated AST for this code would have objects representing tokens in following fashion:



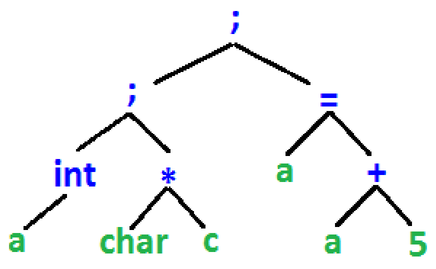
After you traverse this tree, the output should have all the tokens that were there in the input, including the final semicolon. That is, the output should be something like:

int a ;
a = a + 5 ;

Example 2: if your input source code is:

int a; char* c; a=a+5;

The generated AST is:



Then, the output should be something like:

int a ;
char * c ;
a = a + 5 ;