Project 5: Generative Grammars

Project Due: 23rd Apr. at the beginning of class

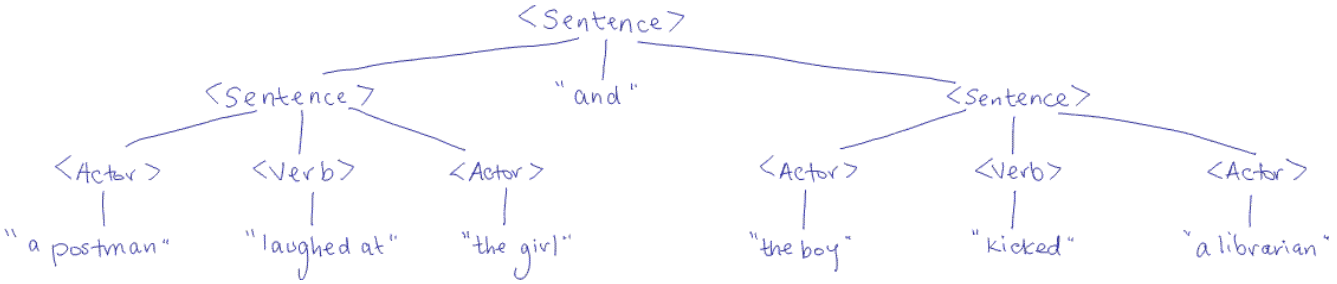
Overview: In this project you are going to implement a program to generate random sentences using a generative grammar.

Goal: The goal of the project is to gain experience with creating and manipulating a hierarchical data structure and with using an STL class.

Description: For this assignment, you are going to implement a program to generate random sentences using a context free grammar. A grammar is a logical specification of the set of valid sentences in a particular language. Context free grammars are used not just in AI and linguistics but also in compilers. A context free grammar is specified using a set of rules made up of terminal and non-terminal symbols. A terminal symbol is an actual word/sub-phrase in the language, e.g. “the boy” “kissed” “green”. A non-terminal symbol is a place-holder in a phrase, e.g. <Noun> <Verb> <Sentence>. The job of a grammar is to specify the rules for transforming non-terminal symbols into terminal symbols. For example, consider the very simple grammar below:

<Start> → <Sentence>
<Sentence> → <Actor> <Verb> <Actor> | <Sentence> and <Sentence>
<Actor> → the boy | the girl | a postman | the librarian
<Verb> → kicked | waved to | laughed at | annoyed

Take for example, the last line of the grammar. It specifies that the non-terminal <Verb> can be replaced with the terminal “kicked” or the terminal “waved to” … (The | symbol separates different potential substitutions.) The grammar should define at least one substitution for every non-terminal in the grammar. The job of a sentence generator is to start with just the non-terminal <Start> and from there perform a series of substitutions to get to a phrase containing all terminals. This is generally represented using a tree, for example:



The grammars are context free because each time the generator comes to the non-terminal <Verb> it will substitute in either “kicked”, “waved to”, “laughed at” or “annoyed” regardless of what substitutions it made earlier or later in the phrase.

In theory, a generative grammar like the one above could create an infinite length sentence: <Sentence> and <Sentence> and <Sentence> and … However, in practice you can usually rely on randomization to make the sentence finish.

For this project you are going to be given an example generative grammar to get you started, but you should be sure to create some of your own for testing (and possibly for entertainment) purposes. The grammars will be stored in .txt files

and you will ask the user what grammar they would like to use in order to generate the sentence. You may not modify the format of the grammar files.

You will start with the code you wrote from project 4 which will allow you to read the grammar rules into a hashtable. You should then use a tree data structure to build your sentence. Your tree will start with a root node holding the non-terminal <Start>. From there, you will recursively traverse the tree and at every node representing a non-terminal symbol, you will ask your hashtable to randomly select one substitution from among the possibilities. For example, you could substitute <Actor> <Verb> <Actor> in for <Sentence>. You will generate 3 children nodes for <Sentence>, one holding <Actor> the next holding <Verb> and the final one holding <Actor>. A node representing a terminal symbol has no children, just a string holding its word/phrase. To generate the sentence you will traverse the tree generating children as you go until all the leaves of the tree are terminal symbols. Then, to print out the sentence you will perform a traversal of your tree printing out all of the terminal symbols. Note: you only need to print out the sentence, not the tree structure.

Your tree will not be a binary tree because there may be more than 2 children for a particular node. Since you don't know how many children each node will have, you should use a Vector (from the STL library) to store the children node pointers.

Your program should offer a simple menu system that allows the user to switch from using one grammar to another (don't forget to empty your hashtable and your tree if this happens) or to generate multiple sentences using the same grammar (don't forget to empty the tree if this happens).

Programming Practice Requirements: There should be separate .h and .cpp files for all the classes in addition to your main program source code file. Each file should be appropriately commented with your name and a description of the file at the top. Each function definition should be described with comments as well. You should also follow all of the standard good programming practices, e.g. meaningful variable names, modular design, etc. As with all programs there should be significant error checking. The program should not accept illegal values. If an illegal value is entered, a message should be given and the user should be prompted for the value again.

You MUST not have any memory leakage in your program, so any dynamically allocated variables / data structures need to be deallocated when they are no longer needed.

The interface to the program should be self-explanatory and should make it easy for the user to generate multiple different sentences and to use different grammars on different sentences. Part of your grade will be determined by how usable the interface to your program is.

It is expected that the code you submit will compile and run. Non-compiling code will earn less than 50% credit. You should also thoroughly test your code to find and eliminate logic errors. It should not be possible to crash your program. Code that crashes due to a logic error will be heavily penalized.

You can earn a couple of points extra credit on the assignment by submitting a substantial and entertaining grammar that you have created yourself. The grammar should be completely specified and in the same format as the sample grammars. Please test your program on the grammar to make sure the grammar doesn't cause a heap overflow exception due to the grammar having too much recursion.

Project Submission: You should email a .zip file containing the whole Visual Studio project to me at ebowring@pacific.edu. Please delete the "debug" subfolder before you zip up your project. If your email provider objects to sending zip files, please bring your code to class on a USB stick.