CS 311 - Programming Language Concepts
Programming Assignment #5

Objectives:

- To compare the following in Java and C++:
  - text file I/O
  - support for data abstraction
  - class libraries
  - random number generation
  - exception handling
  - string manipulation
  - command line arguments
  - dynamic storage allocation
- Operator overloading in C++
- More practice with Context Free Grammars

The Inspiration:

In the past decade or so, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers have also facilitated all sorts of student work from English papers to calculus. One important area of student labor that has been painfully neglected is the task of filling up space in paper and extension requests, etc. with important sounding and somewhat grammatically correct random sequences. An area, which has been neglected, that is, until now...

The “Random Sentence Generator” is a handy and marvelous piece of technology to create random sentences from a context free grammar. There are profoundly useful grammars available to generate extension requests, generic Star Trek plots, your average James Bond movie, "Dear John" letters, and more. You can even create your own grammar. Fun for the whole family! Let’s show you the value of this practical and wonderful tool:

Tactic #1: Wear down the professor's patience.

I need an extension because I used up all my paper and then my dorm burned down and then I didn't know I was in this class and then I lost my mind and then my karma wasn't good last week and on top of that my dog ate my notes and as if that wasn't enough I had to finish my history paper this week and then I had to do laundry and on top of that my karma wasn't good last week and on top of that I just didn't feel like working and then I skied into a tree and then I got stuck in a blizzard at Tahoe and on top of that I had to make up a lot of documentation for

the Navy in a big hurry and as if that wasn't enough I thought I already graduated and as if that wasn't enough I lost my mind and in addition I spent all weekend hung-over and then I had to go to the Winter Olympics this week and on top of that all my pencils broke.

Tactic #2: Plead innocence.

I need an extension because I forgot it would require work and then I didn't know I was in this class.

Tactic #3: Honesty.

I need an extension because I just didn't feel like working.

Your task is to implement RSG in both Java and C++ and then write a comparison of the two languages with respect to the features mentioned in the "Objectives" section above.

Grammar files:

Here is an example of a simple grammar:

```
The Poem grammar
{
<start>
The <object> <verb> tonight;
}

{
<object>
waves;
big yellow flowers;
slugs ;
}
{
<verb>
sigh <adverb> ;
portend like <object>;
die <adverb> ;
}
{
<adverb>
warily ;
grumpily ;
}
```

According to this grammar, two possible poems are "The big yellow flowers sigh warily tonight" and "The slugs portend like waves tonight." Essentially, the strings in brackets (<>) are the *variables*, which expand according to the rules in the grammar. The other strings are the *terminals* in the grammar.

A definition consists of a variable and its set of "productions" each of which is terminated by a semi-colon ';'. There will always be at least one and potentially several productions that are expansions for the variable. A production is just a sequence of words, some of which may be variables. A production can be empty (i.e. just consist of the terminating semi-colon) which makes it possible for a variable to expand to nothing. The entire definition is enclosed in curly braces '{' '}'. The following definition of "<verb>" has three productions:

{
<verb>
sigh **<adverb>** ;
portend like **<object>** ;
die **<adverb>** ;
}

Comments and other irrelevant text may be outside the curly braces and should be ignored. All the components of the input file: braces, words, and semi-colons will be separated from each other by some sort of white space (spaces, tabs, newlines), so you will be able to use those as delimiters when parsing the grammar. And you can discard the white-space delimiter tokens since they are not important. No token will be larger than 128 characters long, so you have an upper bound on the buffer needed when reading a word, however, when you store the words in C++, you should not use such an excessive amount of space, use only what's needed.

Once you have read in the grammar, you will be able to produce random expansions from it. You begin with the single variable <start>. For a variable, consider its definition, which will contain a set of productions. Choose one of the productions at random. Take the words from the chosen production in sequence, (recursively) expanding any, which are themselves variables as you go. For example:

```
<start>
The <object> <verb> tonight.                    --expand
<start>
The big yellow flowers <verb> tonight.      --expand
<object>
The big yellow flowers sigh <adverb> tonight.     --expand
<verb>
The big yellow flowers sigh warily tonight.       --expand
<adverb>
```

Since we are choosing productions at random, doing the derivation a second time might produce a different result and running the entire program again should also result in different patterns.

Design:

You are required to implement `RandomSentenceGenerator` classes with the following methods:

<table>
<tr><th>Java</th><th>C++</th></tr>
<tr><td colspan="2">A constructor that is passed the name of the file containing the grammar rules.</td></tr>
<tr><td colspan="2">A method, <code>randomSentence()</code>, to return a random sentence generated from the grammar.</td></tr>
<tr><td>A <code>toString()</code> method to return the grammar rules in a readable format.</td><td>The overloaded output stream operator << to display the grammar rules in a readable format.</td></tr>
<tr><td></td><td>A destructor to delete any dynamically allocated space used to store the grammar.</td></tr>
</table>

You may add methods to your class as needed. You must implement a separate `main` program to test your `RandomSentenceGenerator` class.

Storing the grammar:

Feel free to use any class library ADTs to store and manipulate the various components of the grammar. In C++, be sure to store the terminals and variables as strings allocated to appropriate size (i.e. do not store using a large fixed-size buffer). Recall that a hash table is particularly good for doing quick lookups, so data you often need to search would best be organized in a hash table, although you may use other appropriate data structures if you choose.

Expanding the grammar:

Once the grammar is loaded up, begin with the `<start>` production and expand it to generate a random sentence. Note that the algorithm to traverse the data structure and print the terminals is extremely recursive.

The grammar will always contain a `<start>` variable to begin the expansion. It will not necessarily be the first definition in the file, but it will always be defined eventually. Your code can assume that the grammar files are syntactically correct

(i.e. have a start definition, have the correct punctuation and format as described above, don't have some sort of endless recursive cycle in the expansion, etc.). The one error condition you should catch reasonably, and for which you should throw an ***exception***, is the case where a variable is used but not defined. It is fine to catch this when expanding the grammar and encountering the undefined variable rather than attempting to check the consistency of the entire grammar while reading it.

The names of variables should be considered case-insensitively, `<NOUN>` matches `<Noun>` and `<noun>`, for example.

Input:

Your program should take one *command line argument*, which is the name of the grammar file to read. An error message should be displayed if the file is not found.

Output:

Your program should print the *grammar rules* and *three random sentences* from the grammar and exit.

Note: When printing the random sentences, each terminal should be preceded by a space except the terminals that begin with punctuation like periods, commas, dashes, etc., which look dumb with leading spaces.