Assignment 2: Random Sentence Generator

Based on Julie Zelenski's (zelenski@cs.stanford.edu) CS107 Assignment. It rocks.

The Inspiration

In the past decade or so, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers also 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 papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences. An area which has been neglected—that is, until now.

Due: Thursday, October 26th at 11:59 p.m.

The Random Sentence Generator is a handy and marvelous piece of technology to create random sentences from a structure known as a **context-free grammar**. A grammar is a template that describes the various combinations of words that can be used to form valid sentences. 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 TA'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 doctoral thesis 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 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.

What is a grammar?

A grammar is just a set of rules for some language, be it English, the C programming language, or an invented language. If you go on to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a contextf-free grammar (**CFG**). 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 variables which expand according to the rules in the grammar.

More precisely, each string in brackets is known as a **non-terminal**. A non-terminal is a placeholder that will expand to another sequence of words when generating a poem. In contrast, a **terminal** is a normal word that is not changed to anything else when expanding the grammar. The name terminal is supposed to conjure up the image that it is a dead-end— no further expansion is possible from here.

A definition consists of a non-terminal and its set of "productions" or "expansions", 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 non-terminal. A production is just a sequence of words, some of which may be non-terminals. A production can be empty (i.e. just consist of the terminating semi-colon) which makes it possible for a non-terminal 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, 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 will always begin with the single non-terminal `<start>`. For a non-terminal, 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 non-terminals 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.

Choosing The Grammar File

<u>Your program should take one argument</u>, which is the name of the grammar file to read. As with all UNIX programs, you can give a full or relative path as an argument; the relative path will save you typing. For example, to read from the `dump.g` grammar file which is the `grammars` subdirectory of the current directory:

```
% rsg grammars/dump.g
```

Your program should create **three random expansions** from the grammar and exit.

Getting Started

The starting project is in the leland directory `/usr/class/cs193d/assignments/hw2`**.** This directory contains a skeleton `rsg.cc` file, code for the `Production` class (which you may augment if you'd like) to get you started and to do most of the annoying file reading, code for the `Definition` class (which you can also augment) which does the rest of the file reading, a `Makefile` that builds the project, and subdirectory of grammar files (files named with the extension "`.g`"). You can also get the starting files via anonymous ftp to `ftp.stanford.edu` (seemingly very slow these days) or from the class Web site `http://cse.stanford.edu/classes/cs193d/`.