Random Sentence Generator

Introduction

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...

The “Random Sentence Generator” is a handy and marvelous piece of technology (that you’re going to build!) to create random sentences from a pattern known as a 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 that is generating excuses for late homework:

Example/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 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 started the assignment too late.*

Background

What is a grammar?

A grammar is just a set of rules for some language, be it English, a programming language, or an invented language. For now, we will introduce to you a particular kind of grammar called a Context Free Grammar (CFG). Here is an example of a simple grammar:

The Poem grammar:

```
{
  "grammarTitle": "Poem Generator",
  "grammarDesc": "A grammar that generates poems. ",
  "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". There will always be at least one and potentially several productions that are expansions for the nonterminal. A production is just a sequence of words, some of which may be non-terminals. A production can be empty (i.e., an empty array) which makes it possible for a non-terminal to expand to nothing.

Expanding the grammar

Once you have read in the grammar, you will be able to produce random expansions from it. You begin with a single non-terminal with the "Start" label. 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 nonterminals as you go.

Begin with a 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.

Modelling the grammar

The grammar model will be provided in a JSON file.

- The JSON file will specifically have 2 keys:
    - "start" which will contain a list of non-terminals to start with ○ "grammarTitle" which is a user-readable name for the grammar.
- The file will optionally include:
    - a "grammarDesc" key, where the value is a user-friendly description of the grammar.
- All other keys will be a non-terminal, with the values being an array of strings with a production for that non-terminal.
- Your code can assume that the grammar files are syntactically correct (i.e. have a start and grammarTitle 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 (regarding the grammar) you should catch reasonably is the case where a non-terminal is used but not defined.
    - It is fine to catch this when expanding the grammar and encountering the undefined non-terminal rather than attempting to check the consistency of the entire grammar while reading it.
- The names of non-terminals should be considered case-insensitively, matching "Adj" and "adj", for example.

Printing the result

Each terminal should be preceded by a space when printed except the terminals that begin with punctuation like periods, comma, dashes, etc. (which look silly with leading spaces).

Choosing the grammar file

Your program should take one argument, which is the name of a folder/directory, where all grammars in that directory are loaded/read in.

The User Interface

In addition to generating random sentences, you will need to build a command-line interface to your system. When the program starts, a directory name will be provided as an argument. This directory will include several grammar files.

Your program will allow the user to choose one of these grammars. When a grammar is chosen, a generated sentence will be displayed. Ask the user if they would like another one, and continue generating and printing sentences until the user says no. Then, go back to the main menu, and allow the user to choose another grammar if they would like.

The entire exchange ***might*** look something like this:

```
./rsg grammars/ Loading
grammars...

The following grammars are available:
1. Insults
2. Term Paper Generator
3. Dear John letter

Which would you like to use? (q to quit)
1

With the fury of Thor's belch, may the hosts of Hades find your
earlobes suddenly delectable.

Would you like another? (y/n) y

You mutilated goat.

Would you like another? (y/n) n

The following grammars are available:
1. Insults
2. Term Paper Generator
3. Dear John letter

Which would you like to use?  (q to quit) q
```

Some tips

- Testing programs with randomization:

    - To ensure that your program is doing what you expect it to do, you can seed the random number generator with a specific number. Then, every time you run the program it will generate the same sequence of numbers, and the output should be predictable.
- Think about the user.
    - Consider the point of view of the user and make it a pleasant experience for them. You may want to have a friend run your program so you can make sure it makes sense to them.
- A couple of grammars are provided for you. Feel free to create your own, specifically maybe one that would be easy to test with.
- Consider if you want to load all the grammars when the program starts up, or when they will be used.
    - Think about how you'll want to utilize memory when possibly loading many large grammars.

Deliverables

- All of your source and test code, no class files or other binaries
- A README.md file, where you describe how to run your program ○ Include the entry point to your program.
    - Give a high-level description of key classes/methods. ○ Include any assumptions you made about the nature of the problem. ○ Include steps you took to ensure correctness. ○ Make sure it's clear how to run your program from the command line and/or from within IntelliJ.
    - If you'd like, you can include some particularly interesting randomly generated sentences you encountered while working on this assignment.
    - Marks will be removed for not having a README.md
- A class diagram representing your code (image format such as pdf, png, jpg)
- Any new grammar files you'd like to share or show off