Final Project

There is a Design Document for this project, due Friday, April 12 at 5pm (must be written and submitted by each programming partner)

The purpose of this assignment is to practice your comprehensive knowledge of data structures and problem solving. The assignment must be completed in pairs.

1. The Problem
   In a video game, the characters often must say phrases throughout the play of the game. The phrases must be of a certain form (e.g., a noun followed by a verb followed by an object). However, the phrases must also be randomized, so that the character does not repeat exactly the same phrase again and again.

   We have been asked to construct a program that randomly generates phrases. The form of each phrase is specified using a input grammar file. (For more information on grammars and how to use them to generate phrases, see the notes from Lecture 25.)

   As an example, consider the input grammar file super_simple.g. (**Note: a .g file is just a plain text file. Open it with a text editor).** A possible random phrase of this form is "The mouse stood on the dog." Another is "The cat sat on the mouse."

   The formatting rules of the input grammer file are very strict, in order to simplify the task of parsing the input.

   - The file contains the definitions of one or more non-terminals.
   - Non-terminals are always denoted with angle brackets, and the name of a non-terminal may not contain any whitespace. For example, `<noun>` and `<long-int>` are correct non-terminals. `<awesome word>` and `<noun >` are not correct non-terminals.
   - Terminals are denoted by the absence of angle brackets.
   - Each non-terminal definition begins with an opening curly brace (on its own

NAVIGATION

Home
- My home
  - Site pages
  - My profile
  - My courses
    - Computer Science
      - Previous Semester
        - CS 1410-1-S13
        - CS_2100_S_13
        - CS2420-S13
          - Participants
          - General
          - Getting started; Java review
          - Generic programming; Object Oriented Programming
          - Algorithm analysis; Data Structures
          - Basic Sorting Algorithms
          - Recursive Sorting Algorithms
          - Linked Lists
          - Stacks and Queues

line with no extraneous spaces). The non-terminal being defined appears alone on the next line. The choice of one or more productions to which the non-terminal can be expanded appear next (one per line). A single production consists of one or more terminals and/or non-terminals. A single blank space may appear between each terminal and non-terminal; however, no extraneous spaces appear before the first (non-)terminal or after the last (non-)terminal. Finally, the non-terminal definition ends with a closing curly brace (on its own line with no extraneous spaces).
- Every non-terminal used in a production is defined somewhere in the file, but not necessarily before (or after) the production in which it is used.
- The starting non-terminal is always identified as `<start>`.
- Comments may appear in between non-terminal definitions. Therefore, any lines outside of the curly braces should be ignored.
- Error-checking on the input grammar file is not required, and you may assume that all grammar file used for grading will be correct.

The following are some sample input grammar files and possible random phrases for each.

- poetic_sentence.g
  The waves portend like big yellow flowers tonight.

- mathematical_expression.g
  (y + ((x - 2) * e))

- assignment_extention_request.g
  I need an extension because I had to practice for an alligator wrestling meet, and as if that wasn't enough I just didn't feel like working, and then, just when my mojo was getting back on its feet, my dorm burned down.

2. Requirements

- Create a new class called `RandomPhraseGenerator` in a new package called `comprehensive`. The user will start your program by running this class (i.e., by calling `RandomPhraseGenerator`'s main method with command line arguments).
- You may create as many other new classes as needed. Make sure that all Java files required by your program are in the`comprehensive` package.
- The path and name of the input grammar file will be given as input to your program via the command line (as the first command-line argument). Also given as input will be the number of random phrases your program should

generate (as the second command-line argument).
- The output of your program is simply the random phrase(s), printed one per line. The output must preseve any blank spaces that do or do not appear between terminials and non-terminals. For example, given input grammar file super_simple.g, "The cat sat on the mouse." is a possible random phrase, but "The cat sat on the mouse ." is not.
- When expanding a non-terminal, each of its production rules must have an equal probability of being selected.
- Because all grading will occur outside of Eclipse, you **must** make certain that your program runs correctly in the terminal of a CADE lab machine before submission. As an example, to run your program such that it generates five random phrases for the input grammar file poetic_sentence.g, issue the following command.

  ```
  java comprehensive/RandomPhraseGenerator poetic_sentence.g 5
  ```

  Your working directory must contain both the input grammar file and your `comprehensive` package.
- Command-line arguments can be specified in Eclipse, using the *Run Configurations* menu.
- You may assume that your program will not be tested for input grammar files that are non-existent or incorrectly formatted.

NOTE: It is intentional that you are being given no guidance as to how to solve the problem of generating random phrases. It is critical that you gain experience solving a problem "from scratch," designing the structure of your classes and methods, as well as, choosing the best data structures and algorithms for the problem. Because the readers of your code will have no assumptions about how it is organized, you **must** document it well. Finally, if you are feeling overwhelmed by the problem and are not sure how to get started, see these steps as a suggestion. *Try to avoid looking at the suggestion*, and instead, come up with your own strategy for how to attack the problem.

3. Design Document (must be written and submitted by each programming partner) ***due Friday, April 12 at 5pm***
4. When preliminary coding is complete and your program compiles without error or warning, test the program thoroughly and systematically.

   Consider thousand.g as an example of a grammar designed to stress at least one aspect of your solution.

Your code should be well-commented (Javadoc comments are recommended) and formatted such that it is clear and easy to read. Be sure to put the names of both programming partners in the header comment of each file.

Zip your source code files (no `.class`, `.java` only) and **upload the zip file here by 5pm on Wednesday, April 24**. Please submit just one solution per pair (i.e., one partner should upload the zip file, the other should not upload anything).

5. Analysis Document (must be written and submitted by each programming partner) ***due Wednesday, April 24 at 11:55pm***

Submission status

Submission status	Submitted for grading
Grading status	Graded
Due date	Wednesday, 24 April 2013, 11:55 PM
Time remaining	Assignment is overdue by: 81 days 14 hours
Last modified	Wednesday, 24 April 2013, 9:54 PM
File submissions	analysis final.pdf final graph.xlsx

Feedback

Grade	81.00 / 85.00
Graded on	Sunday, 5 May 2013, 12:04 AM
Graded by	

	 Sarah Hong
Feedback comments	Tests Passed - 115/115 Phrase Points - 50/50 Testing Points - 3/3 Style Points - 7/7 Analysis Points - 21/25 Total Points - 81/85
Feedback files	barsketis.pdf