Optional Lab 5

Due 12 March

Handout 6
CSCI 136: Spring, 2007
7 March

This lab is optional. You may turn it in by next Monday if you wish to recieve some extra credit. Other options for this week include:

1. Working on the extra credit problems from the Recursion lab.
2. Writing the lab at the end of Chapter 8. (The starter file `LinkedList.java` can be found at `/usr/cs-mac-local/share/cs136/labs/list/LinkedList.java`.)
3. Writing any other program you find interesting. Come talk to me if you want to try to explore a different problem of your own choosing.

Random Sentence Generator

The goals of this lab are to

1. learn about context-free grammars,
2. modify and extend an existing code base, and
3. produce fairly entertaining, but also grammatically correct, random text.

The main shortcoming of the frequency-based random text generator from Lab 2 is that even high levels of analysis on large volumes of text still doesn't guarantee that the output is well-formed English. In this lab we explore a more structured way to generate random text.

Specifically, the "Random Sentence Generator" creates 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. I will provide you with grammars to generate extension requests, poems, and more. You can even create your own. Examples of the output for generating homework extension requests:

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 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 on Mt. Greylock 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.

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

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

This assignment dates back to all the way to the '80's. It was originally designed by Mike Claron and Julie Zelenski at Stanford, and it was the first program that I wrote in college— I got 14.5 out of 20 points on it...

1 What is a Grammar?

A grammar is a set of rules for some language, be it English, the Java 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 Context Free Grammar (CFG). Here is an example of a simple grammar to generate poems:

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

<object> {
     waves                ;
     big yellow flowers   ;
     slugs                ;
}

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

<adverb> {
     warily             ;
     grumpily           ;
}
```

Here are two possible poems generated by this grammar:

```
    The big yellow flowers sigh warily tonight

    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 list of productions is enclosed in curly braces '{' '}'. The following definition of `<verb>` has three productions:

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

There will always be whitespace surrounding semi-colons and braces to make parsing easy. I will provide the code to read in the grammar file. Your job will be to put that information in a data structure representing the grammar.

Once you create the grammar data structure, you will be able to produce random expansions from it. You 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.

2 Data Structure and Algorithm Design

The data structures to store a grammar and create sentences are divided into three classes. The javadoc documentation for them appears on the handouts web page. Refer to them for a complete description of their methods and how to use them.

- `Definition`: A `Definition` stores the productions for a single non-terminal. Each production is simply a `Vector` of `Strings` that may be either terminals or non-terminals.
- `DefinitionTable`: This class allows you to keep track of `String`-`Definition` associations.
- `RandomSentenceGenerator`: You will need to modify this class to create `Definition`s for the grammar and insert them into the `DefinitionTable`.

  Some of the code to parse the file is provided– you will need to add additional code to create productions and definitions appropriately.

  Once the grammar is properly created, you should implement an `expandWord` method. This method takes a word as an argument. If the word is a non-terminal like "`<start>`", it gets a random production for that non-terminal and tries to recursively expand each word in that production. If the word is a terminal, it is just printed.

  You should not need to change the existing code in `Definition` or `DefinitionTable`, but feel free to add reasonable additional features to complete the assignment.

You should familiarize yourself with all of these classes before you start writing any code.

3 Notes

1. Start by copying the starter directory with the following command:

   ```
   cp -r /usr/mac-cs-local/share/cs136/labs/rsg .
   ```

2. The program reads from the terminal, so run the program with a command like

   ```
   java RandomSentenceGenerator < Poem.g
   ```

3. The grammar will always contain a `<start>` non-terminal 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 (ie, with an

assertion) is the case where a non-terminal is used but not defined. It is fine to catch this when expanding the grammar rather than attempting to check the consistency of the entire grammar while reading it.

4. When generating the output, you do not need to store the result in some intermediate data-structure– just print the terminals as you expand. Each terminal should be preceded by a space when printed except the terminals that begin with punctuation like periods, comma, dashes, etc. You can use the `Character.isLetterOrDigit` method to check whether a character is punctuation mark. This rule about leading spaces is just a rough heuristic, because some punctuation (quotes for example) might look better with spaces. Don't worry about the minor details, we're looking for something simple that is right most of the time and it's okay if is little off for some cases.

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

4 Submission

Submit your program (and any grammar files you create) using the `turnin` utility no later than 11:59pm on the due date.

As in all labs, you will be graded on design, documentation, style, and correctness. Be sure to document your program with appropriate comments, including a general description at the top of the file, a description of each method with pre- and post-conditions where appropriate. Also use comments and descriptive variable names to clarify sections of the code which may not be clear to someone trying to understand it.