Lab 1: Recursion

Introduction

Tracery (tracery.io) is a simple text-expansion language made by one of your TAs as a homework assignment for one of Prof. Mateas's previous courses. It now has tens of thousands of users, and runs about 7000 chatbots on Twitter (you never know where a homework will take you!).

Tracery uses context-free grammars (https://en.wikipedia.org/wiki/Context-free_grammar) to store information about how to expand rules. A Tracery grammar is a set of keys, each of which has some set of expansions that can replace it. In our version, the line `beverage:tea|coffee|cola|milk` means that the *symbol* `beverage` can be replaced with any of those four options. You *replace* a symbol whenever you see it in hashtags. This rule `#name# drank a glass of #beverage#` will have the `name` and `beverage` symbols replaced with expansions associated with those symbols. In the case of the `beverage` rule above, which has four possible expansions, one will be picked at random. If the replacement rule *also* has hashtags in it, we replace those, and if those replacements have hashtags.... we keep replacing things until all the hashtags are gone, *recursively*.

In this assignment, you will be implementing a simplified version of Tracery in Java, and then using it to generate generative text. You will also be writing your own grammar to generate new texts (hipster cocktails, emoji stories, or nonsense poems).

Outline

- Compile and run a Java program
- Save all the arguments
- Load the Tracery files
- Output all the rules
- Expand rules and print them to the screen

Compile and run a Java program

This program has several files (`Rule.java` , `RuleSection.java` , and `TraceryRecursion.java`). We can't *run* these files as code, as we would with other "interpreted" languages (like Javascript or Python). For Java, we need the computer to put them all together and translate it to machine code, in a process called "compiling".

You will compile and run your Java program from the command line. When you see `$` , this means that it is a command that you will enter in the *command line*. Windows and Unix (such as Linux or the Mac terminal) command lines are a little different, be sure you know the basics of navigating the one you are using.

Do you have Java installed on your machine? What version? Let's find out! Type the following into your command line: `$ javac -version` `$ java -version` You should have at least Javac 1.8 and Java 1.8 (often called "Java 8") installed. If that's not the case, this is a good time to fix that by updating your Java. We will be using some features of Java 8 in this class.

Compile your java program. `$ javac TraceryRecursion.java` So far, it will compile without errors.
Look in the folder, and you will see that you have a new file `TraceryRecursion.class` . This is the **compiled** version of your file. You can now *run* this file by typing: `$ java TraceryRecursion` You should get the output `Running TraceryRecursion...` .

Try typing `javac TraceryRecursion` or `java TraceryRecursion.java` . Both of these should give your errors. Compiling requires the `.java` extension, and running requires *not* having it. But its easy and common to misremember that, so look at what there errors look like, so that when it happens later, you know what caused it.

You can compile and run in separate steps. But sometimes you want to compile and run with one line, and you can combine them like this: `$ javac TraceryRecursion.java && java TraceryRecursion`.

Store arguments (Java refresher)

Many java programs need outside information provided by the user. This lets users change the behavior of the program (or tell it which data to use) without having to modify the source code.

The `main` method of `TraceryRecursion.java` will be the first method called when running the program. This method is `static`, meaning that the method is associated directly with a class and can thus be called without creating an object that is an instance of the class, and its return type is `void`, meaning that it won't return anything. It also has **parameters**, an array of Strings called `args` (you could call this parameter anything, but the tradition is to call it `args` or `arguments`).

Our program will take 0-4 arguments:

- the name of the file containing the grammar (e.g.: "grammar.txt")
- the starting symbol to begin the grammar expansion (e.g.: "#story#")
- the number of times we want to perform that expansion (e.g: "5")
- a seed value for the random number generator (e.g: "123") (Seeded random values allow us to make the random generator *predictable*)

You can see that in `main`, each variable is given a default value. For example, if no seed value is given in the arguments, the seed value will be the current time (this means you will get different random results each time you run the program, unless a specific seed is specified in the arguments).

TODO #1

For each of these parameters, test whether enough arguments have been specified to set that paramter. For example, to set the grammarFile parameter there would have to be at least one argument given. You can test the length of the `args` array to determine how many arguments have been passed in. Override each of these default parameters with its argument override. For the numbers, you will need to use `Integer.parseInt` and `Long.parseLong` to turn the arguments from Strings into numbers. **NOTE: Do not add or remove code outside of the START and END comments. Keeping all your code between those two comments will help us grade your work.**

Load a Tracery file

Java has many helpful built-in data structures. Lists, Arrays, and Hashtables are only a few of them. In this assignment, we will be loading and storing a Tracery grammar as a Hashtable. The Hashtable has a list of *keys*, and each key has some data associated with it, much like a dictionary stores words with their definition. Hashtables come with some helpful methods included:

- `table.get(String s)` returns the data for that key, but it will throw an error if that key is not in the table.
- `table.put(String s, DATA)` creates an entry for that key (or replaces it) and stores the data there.

In this part of the assignment, we will implement `loadGrammar`, which takes a String representing a file path, then **loads a file**, parses it (breaking it into sections), and stores it in our grammar object.

In the `loadGrammar` method, we have already added the line that creates the Hashtable. You can see that the data type has a weird ending `<String, Rule[]>`, which tells Java that this Hashtable uses *Strings* to look up *arrays of Rules*, so if we try to use numbers as look up values, or store Strings instead of arrays of Rules, Java knows to throw an error.

We also included a basic way to load a file. Loading files might cause an IO exception (for example, if the file does not exist), so for complex reasons, we need to have the loading line (`new String(Files.readAllBytes(Paths.get(path)))`) wrapped in a `try/catch` block. We will talk more about throwing and catching exceptions later this quarter. For right now, this loads a whole file as a string. By appending `.split('\\r?\\n')` on the end, we take the resulting string and splits it into an array of lines.

TODO #2

For each line in the array of lines, store the data in the `grammar` hashtable. You can use either a familiar `for (int i = 0...)` loop, or use the newer `for (String line: lines)` loop, if you don't need to see the index value.

We used the `split` String method above to split apart the file into lines. Use `split` again, but splitting on the character `:`. Everything before the `:` is the key. Store it in a variable named `key`. Everything after it is the rule expansion(s).

First split apart the rules into an array of Strings. For each line in the file, this will give you a String array of length 2, where the first element of the array is an expandable symbol (which will be a `key` in the `grammar` hashtable), and the second element of the array is the expansion(s) for this symbol. But there is possibly more than one expansion for a symbol. If you look at the grammar files, you can see where there is more than one expansion for a symbol when there's a comma-delimited list of expansions after a symbol. So we have to further split the expansions into an array of Strings, one for each expansion (by splitting on `,`).

Once you have an array of Strings (of expansions) we have to turn it into an array of Rules. Initialize an array, `Rule[]`, of the right size. Use another loop (inside the loop iterating through lines) to iterate through your array of strings and, for each expansion, create a Rule (it takes a single string as a parameter), and add it to your array of Rules. You now have the right kind of data (`Rule[]`) to store in the hashtable.

Store your data in `grammar`. What was that method that stores data in a grammar (hint: we describe it above)?

In the `main()` method there's already a call to `outputGrammar`; this prints your grammar to the console, so you can verify that it loaded it correctly (that the correct expansions are associated with the correct symbols).

Implementing recursion

Now that you have successfully loaded a file and stored it as a `Hashtable<String, Rule[]>` grammar, we can implement the method that generates the generative text! Open up `Rule.java`, and look at the constructor for this class (`Rule(String raw) {`). This constructor takes the string that you passed to it in **TODO #2** and splits it into sections (using "#"). The **even** indexed entries of `sections` are plain text (ie. the zeroth, second, fourth, etc.), and the **odd** indexed entries of the variable `sections` are symbols for expanding (ie. the first, third, fifth, etc.). This might seem a bit confusing at first. In an expansion like `once #character# and #character# went to a #place#` it's clear that splitting on `#` will cause the zeroth, second and fourth elements (the even elements) to be plain text (i.e. `once`, `and` and `went to a`) and the first, third and fifth elements (the odd elements) to be symbols (i.e. `#character`, `#character`, `#place`). But consider the expansion `#name# the #adjective# #animal#`. Won't this even and odd approach get reversed, since the expansion starts with a symbol? It turns out it won't because, if there's a symbol at the beginning, or two symbols right next to each other, splitting on `#` produces **two** strings, the empty string (which is what is before the `#` and then the string with the symbol (which is what is after the `#`, minus the closing `#` since we're splitting on `#`). So splitting this expansion on `#` will produce the following array:

```
sections[0] == ""
sections[1] == "name"
sections[2] == "the"
sections[3] == "adjective"
sections[4] == ""
sections[5] == "animal"
```

So the even and odd relationship still works out, it's just that two of our even entries are the empty string (which is still plain text), which, when we include it in the output of the expansion, won't be visible, and so won't cause us any problems. Phew! Now that we've explained that we can get back to actually writing the code in `Rule.java` to expand the text.

The method `expand` takes a grammar (the type is `Hashtable<String, Rule[]>`) as an argument, and returns the expanded String. But right now, it only returns a copy of its original expansion (with any symbols not expanded). We want it to instead **recursively expand** the odd sections.

TODO #3

Create an array of Strings named `results` that is the same size as the rule sections (`sections.length`). This is where we will store the results of expanding each section. Create a for-loop that iterates down the sections array. Since we need to know whether each section is odd or even, you will need to use `for (int i = 0...)` , rather than `for (String section: sections)` style loops. For each section, if it is *even* (`i%2 == 0`), copy the text from the rule section into the results array.

If it is odd, then this is a symbol that we need to expand. Use the grammar to find the array of expansions for this symbol. We need to pick a "random" index in the array, but we will use the seeded random number generator (so we can reproduce our results using the same random seed). You can get a random int with `random.nextInt(int bound)` . This will return a random integer between 0 (inclusive) and the `bound` argument you pass in (exclusive). Since you're randomly picking from an array of possible expansions, what's the largest random value you will want? Pick out an expansion, and save it as the variable `Rule selectedExpansion` .

We want to store that new value in our `results` array, but since is is an expansion string we need to process it first because it might contain symbols that need to be expanded as well. What method should you call on `selectedExpansion` to do that? Store the resulting text in the results array.

We now have `results` values for each section in this rule! Use `String.join("", results);` to join all of the results together into a single string (with no spaces between them), and return the results of that operation.

Finishing up

Run your program. It should now output however many outputs of the text you want (as specified by the `count` parameter). Try it with and without a seed value. Does it always output something new? Try out a few other grammars, we included `grammar-recipe.txt` which makes bad recipes, and `grammar-emojistory.txt` which tells stories with emoji.

Now edit grammar-yourgrammar.txt. Create something interesting (of about equal complexity to `grammar-story`). We won't judge you on what it makes, consider this a freeplay exercise. Some inspirational twitterbots include @unicode_garden, @DUNSONnDRAGGAN, @orcish_insults, @GameIdeaGarden , @indulgine, @CombinationBot, @thinkpiecebot, @infinite_scream, FartingEmoji.

Make sure your grammar generates without causing errors! We didn't implement a graceful way to recover from bad formatting or missing symbols, so these grammars are brittle, they break easily.

Turn in:

Wow, you're done! Congrats on finishing your first 12B assignment. Zip your folder, and submit it to Canvas.

Checklist:

- Works for no command-line arguments, a few, or all of them
- If a random seed is specified, the output stays consistent. If not, it changes
- If you switch the grammar file, you get output from a different grammar
- You have edited `grammar-yourgrammar.txt` and it runs.

Not sure if its right? To check, see if your output matches this (for these command-line arguments):

```
Running TraceryRecursion...
        with grammar:'grammar-story.txt'        startrule:'#origin#'    seed:10         count:5
Set seed 10

GRAMMAR:
adjective:            "#color#","#emotion#",
place:                "school","the beach","the zoo","Burning Man",
emotion:              "happy","sad","elated","curious","sleepy",
origin:               "once #character# and #character# went to #place#",
color:                "red","green","blue",
name:                 "emily","luis","otavio","anna","charlie",
character:            "#name# the #adjective# #animal",
animal:               "cat","emu","okapi",
once luis the red okapi and luis the sad emu went to the zoo
once anna the red emu and emily the blue emu went to school
```

```
once charlie the blue cat and anna the blue okapi went to school
once charlie the red cat and anna the red emu went to the beach
once anna the sad okapi and anna the sleepy okapi went to the zoo
```

Additional Resources

- How should I name things in Java? https://www.javatpoint.com/java-naming-conventions
- Basic Java Syntax https://www.tutorialspoint.com/java/java_basic_syntax.htm
- Need some in-depth Java tutorials? There are free courses online: https://www.codecademy.com/learn/learn-java, https://www.udemy.com/java-tutorial/
- Command line basics: https://hellowebbooks.com/learn-command-line/
- Get Linux/OSX style command line tools on Windows with Cygwin https://cygwin.com/install.html