Comprehensive Project: Random Phrase Generator

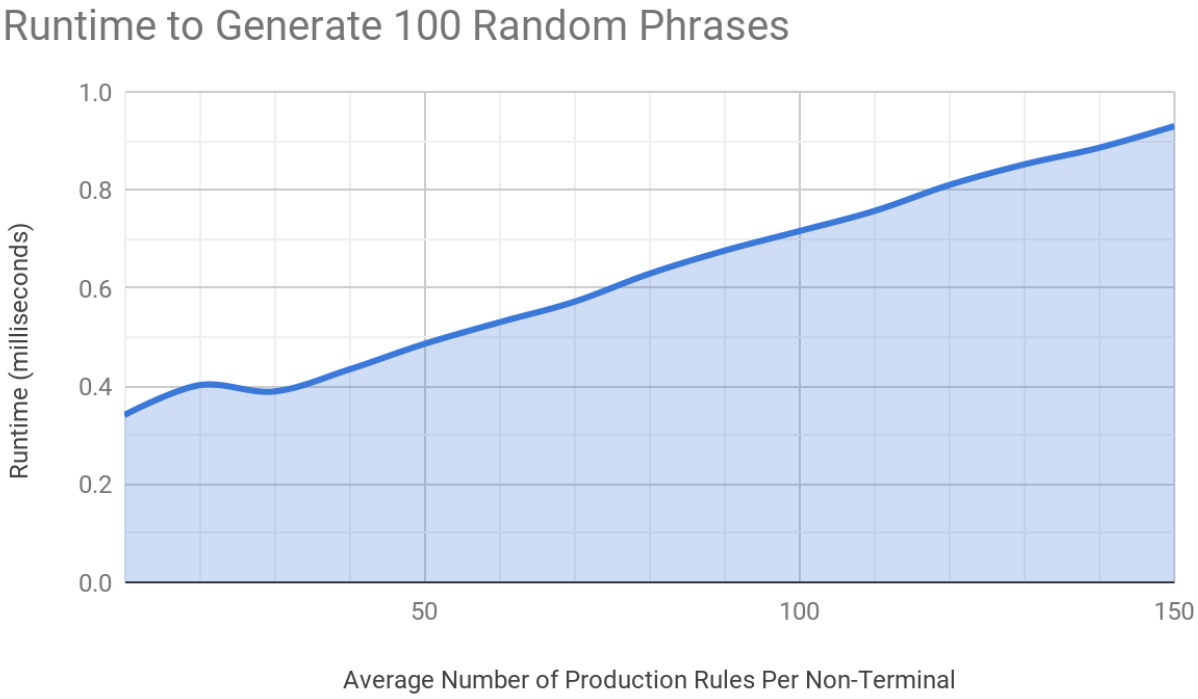
Analysis Document by Qianlang Chen (u1172983)

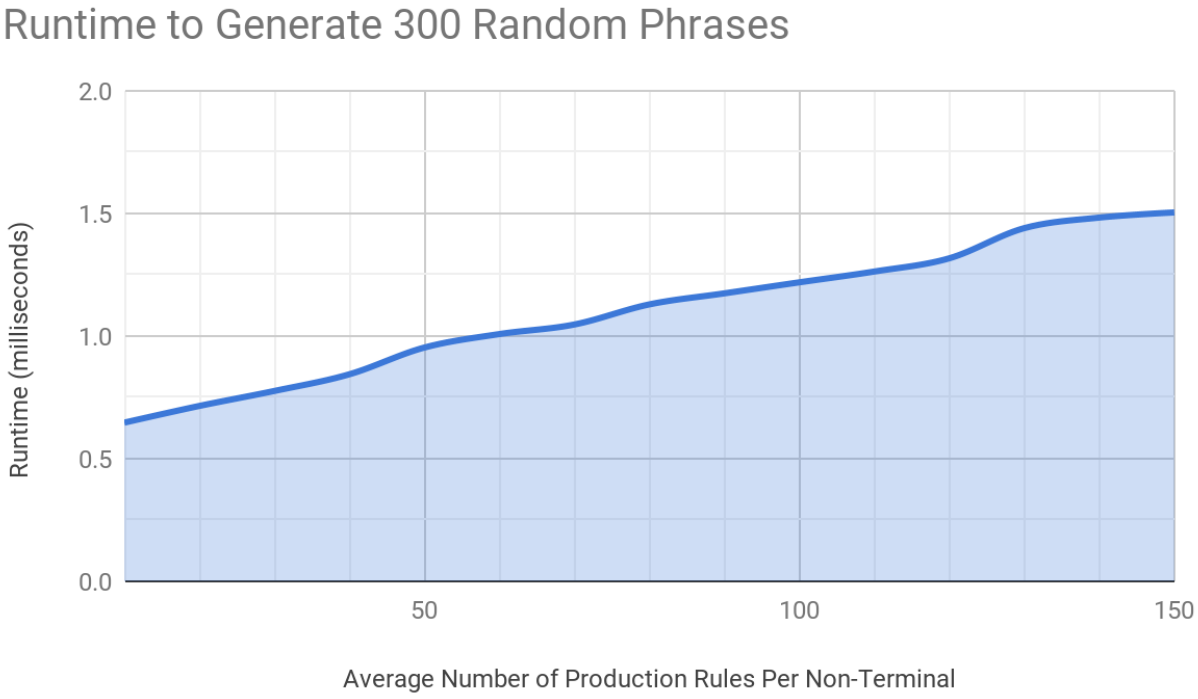
As in the previous 5 programming assignments, I gladly had Kevin Song as my programming partner in this comprehensive project. Kevin did a great job at understanding the problem as well as coming up with proper solutions to solving it. I would definitely choose to work with him again if I had another opportunity.

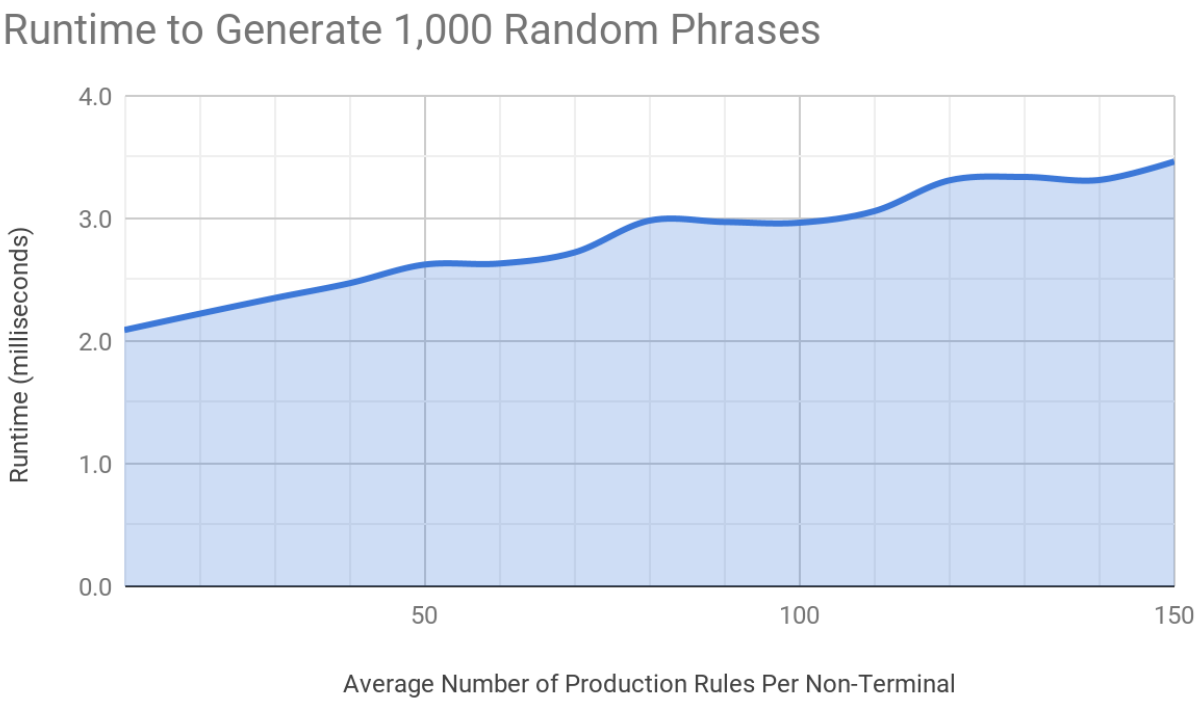
Our solution to this problem was fairly simple and had two major phases: reading and parsing the grammar definition file, then generating a number of random phrases according to the grammar read in. We made an interface called *Text*, to represent a piece of text in a production rule, and it had a *getText()* method, which simply printed the content of the *Text* into the console. We then made two classes to represent terminals and non-terminals in a production rule, and they implemented the *Text* interface. The *Terminal* class's *getText()* method just printed the content of the *Terminal*. The *NonTerminal* class carried a list of production rules, with each rule being a list of *Texts*. When the *NonTerminal* class's *getText()* method was called, a rule is randomly picked from its fellow rule-list, and each Text's *getText()* method is called, making the *getText()* method recursive. Following this idea and structure, the reading phase of our solution read in the file line-by-line and split each production rule into a list of *Terminal* and *NonTerminal* objects. We used Java's *HashMap* class to store the *NonTerminal* instances, with the names of the non-terminals being the keys, so we could find them easily and efficiently. The list of production rules, along with each production rule which is essentially a list of terminals and non-terminals, are both implemented using Java's *ArrayList* class, since it gives a constant running time for look-up. We also considered and tested using a *LinkedList* for each production rule, but it was not as fast as using *ArrayLists*, so we discarded using it.

To better understand the runtime efficiency of our *RandomPhraseGenerator*, we performed two timing experiments. Our first experiment involved how the amount of production rules non-terminals have affects the runtime of our program. We created a number of replicas of

the *"assignment_extension_request.g"* file with different average numbers of production rules per non-terminal, and generated different numbers of random phrases from each grammar file. Here are our results:

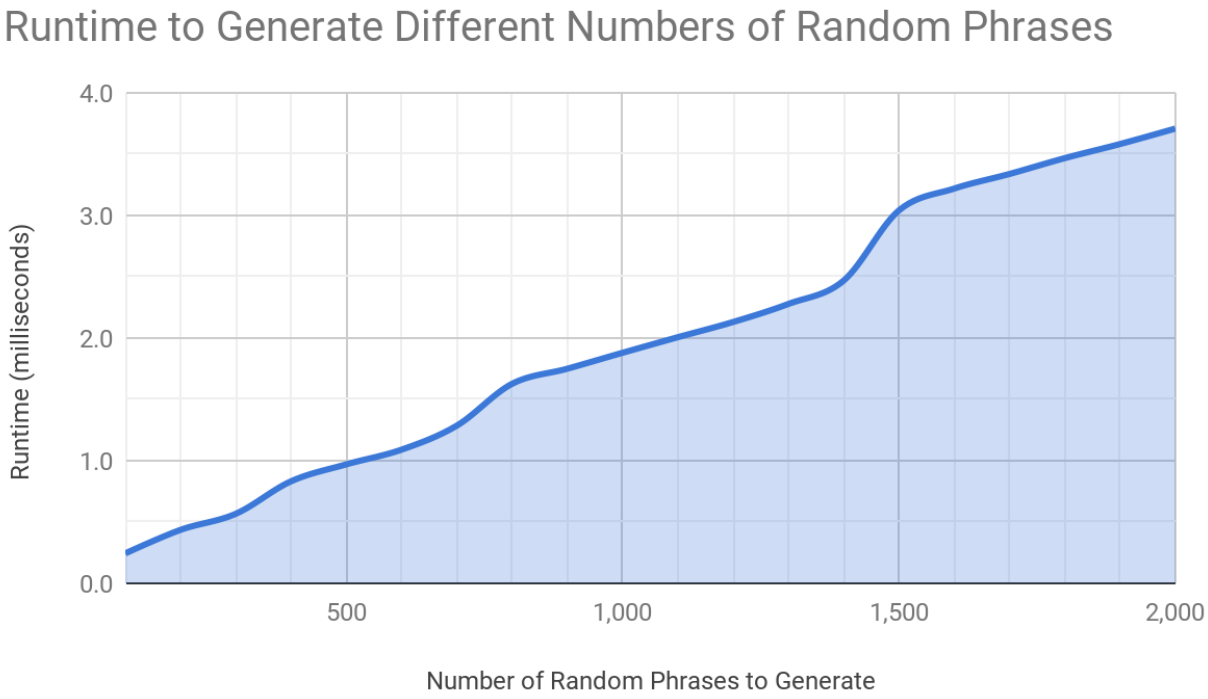






Our results show that the runtime behavior of our *RandomPhraseGenerator* is approximately **linear to the average number of rules a non-terminal has** in the grammar definition file, which matched our prediction: the longer the grammar file is, the longer it takes to process the stuff read in, proportional to the length of the file. However, since the program only reads in the grammar file once, this "linear" running time should have a small impact on the overall runtime, as the number of random phrases to generate gets big. This is demonstrated in the graphs as well: as the number of phrases gets big, the line "flattens out" and grows slower.

We did one more experiment to see how the number of random phrases to generate affects the runtime. We timed our program generating various numbers of random phrases using the same *"assignment_extension_request.g"* grammar file, and here are our results:



The graph shows that our program's runtime behavior is **linear to the number of random phrases to generate**, which also matches our predictions since the more random phrases requested, the longer it takes. The graph shows another interesting thing: the line has several "bumps", and they occur in such a predictable way. My interpretation of this is that the *System.out* buffer is just like an *ArrayList* which requires resizing when filled. I also thought that if we understood more about the *System.out* buffer better, we might have some other ideas to make the printing part of our program more efficient.

According to our results and our implementation techniques, this random phrase generator should have an excellent good runtime efficiency. We used *HashMap* to store the dictionary of non-terminals, which has a constant runtime of adding and looking-up. A recursive method was used to generate phrases, which was essentially a *Stack* and had a constant runtime of adding and removing. The only part that did not have a constant runtime was the traversal of production rule-lists, which are stored using *ArrayLists* and had linear runtime, but linear was as best as we could do since every element must be visited once in order. Overall, our program

should have an excellent performance in terms of its runtime behavior, however, it might not be at its maximum potential since the runtime behavior was not the whole story. Our program was well-written, but there might still be some implementation details that we, as the students, might have left off, making the program not as good as it could be. We might be able to do better if we knew more about the programming language and were more experienced and skilled in programming and problem-solving.