CS106X Autumn 2015

Cynthia Lee

Assignment 2: ADTs

Inspiration credit goes out to Mike Cleron from Google (random sentence generator) and Owen Astrachan from Duke University (word ladder).

Assignment handout authors: Cynthia Lee, Marty Stepp, Julie Zelenski and Jerry Cain.

Due: Friday, October 9th at 5:00pm

Pair programming is permitted on this assignment. See course information sheet and honor code.

Now that you've ingested the CS106 container classes, it's time to put these objects to use. In your role as client, the low-level details have already been dealt with and locked away as top secret so you can focus your attention on solving more interesting problems. Having a library of well-designed and debugged classes vastly extends the range of tasks you can easily take on. Your next assignment has you write three short client programs that heavily leverage the standard classes to do nifty things. The tasks may sound a little daunting at first, but given the power tools in your arsenal, each requires less than a hundred lines of code. Let's hear it for abstraction!

The assignment has several purposes:

1. To stress the notion of abstraction as a mechanism for managing data and providing functionality without revealing the internal representation.
2. To become more familiar with C++ class templates.
3. To gain practice with classic data structures.

This assignment has two projects. The first project is a fun jumble of Stack and Queue ADTs. The second project is really two projects—where only the first is required—and they are grouped together because they are two variations on the same theme (to be explained below). Both parts of the second project illustrate the wonderfully useful and easy to use Map ADT.

PROJECT 1: WORD LADDER [by Julie Zelenski]

A word ladder is a connection from one word to another formed by changing one letter at a time with the constraint that at each step the sequence of letters still forms a valid word. For example, here is a word ladder connecting the word "code" to the word "data". Each changed letter is underlined as an illustration:

code → c<u>a</u>de → ca<u>t</u>e → <u>d</u>ate → dat<u>a</u>

There are many other word ladders that connect these two words, but this one is the shortest. That is, there might be others of the same length, but none with fewer steps than this one. In the first part of this assignment, write a program that repeatedly prompts the user for two words and finds a minimum-length ladder between the words. You must use the **Stack** and **Queue** collections from Chapter 5, along with following a particular provided algorithm to find the shortest word ladder. This part is simpler than Part B.

Here is an example log of interaction between your program and the user (with console input underlined):

```
Welcome to CS 106B Word Ladder.
Please give me two English words, and I will change the
first into the second by changing one letter at a time.

Dictionary file name? dictionary.txt

Word #1 (or Enter to quit): code
Word #2 (or Enter to quit): data
A ladder from data back to code:
data date cate cade code

Word #1 (or Enter to quit):
Have a nice day.
```

Part A Word Ladder sample log of interaction (see course web site for more logs)

Notice that the word ladder prints out in reverse order, from the second word back to the first. If there are multiple valid word ladders of the same length between a given starting and ending word, your program would not need to generate exactly the ladder shown in this log, but you must generate one of minimum length.

Your code should ignore case; in other words, the user should be able to type uppercase, lowercase, mixed case, etc. words and the ladders should still be found and displayed in lowercase. You should also check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, you should check that both words typed by the user are valid words found in the dictionary, that they are the same length, and that they are not the same word. If invalid input occurs, your program should print an error message and re-prompt the user. See the logs of execution on the course web site for examples of proper program output for such cases.

You will need to keep a dictionary of all English words. We provide a file dictionary.txt that contains these words, one per line. The file's contents look something like the following (abbreviated by ... in the middle in this spec):

```
aa
aah
aahed
...
zyzzyvas
zzz
zzzs
```

Your program should prompt the user to enter a dictionary file name and use that file as the source of the English words. If the user types a file name that does not exist, reprompt them (see the second execution log on the next page). Read the file a single time in your program, and choose an efficient collection to store and look up words. Note that you should not ever need to loop over the dictionary as part of solving this problem.

Finding a word ladder is a specific instance of a shortest-path problem of finding a path from a start position to a goal. Shortest-path problems come up in routing Internet packets, comparing gene mutations, and so on. The strategy we will use for finding a shortest path is called breadth-

first search ("BFS"—more on this later in the quarter!), a search process that expands out from a start position, considering all possibilities that are one step away, then two steps away, and so on, until a solution is found. BFS guarantees that the first solution you find will be as short as any other.

For word ladders, start by examining ladders that are one step away from the original word, where only one letter is changed. Then check all ladders that are two steps away, where two letters have been changed. Then three, four, etc. We implement the breadth-first algorithm using a queue to store partial ladders that represent possibilities to explore. Each partial ladder is a stack, which means that your overall collection is a queue of stacks. Here is a partial **pseudocode** description of the algorithm to solve the word-ladder problem:

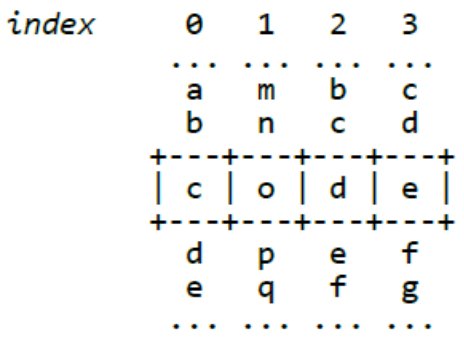
```
finding a word ladder between words w1 and w2:
    create an empty queue of stacks.
    create/add a stack containing {w1} to the queue.
    while the queue is not empty:
        dequeue the partial-ladder stack from the front of the queue.
        for each valid English word that is a "neighbor" (differs by 1 letter)
        of the word on top of the stack:
            if that neighbor word has not already been used in a ladder before:
                if the neighbor word is w2:
                    hooray!  we have found a solution.
                otherwise:
                    create a copy of the current partial-ladder stack.
                    put the neighbor word on top of the copy stack.
                    add the copy stack to the end of the queue.
```

Some of the pseudocode corresponds almost one-to-one with actual C++ code. One part that is more abstract is the part that instructs you to examine each "neighbor" of a given word. A neighbor of a given word w is a word of the same length as w that differs by exactly 1 letter from w. For example, "date" and "data" are neighbors.

It is not appropriate to look for neighbors by looping over the entire dictionary every time; this is way too slow. To find all neighbors of a given word, use two nested loops: one that goes through each character index in the word, and one that loops through the letters of the alphabet from a-z, replacing the character in that index position with each of the 26 letters in turn. For example, when examining neighbors of "date", you'd try:

- aate, bate, cate, ..., zate ← possible neighbors changing 1st char
- date, dbte, dcte, ..., dzte ← possible neighbors changing 2nd char
- daae, dabe, dace, ..., daze ← possible neighbors changing 3rd char
- data, datb, datc, ..., datz ← possible neighbors changing 4th char

Note that many of the possible words along the way (aate, dbte, datz, etc.) are not valid English words. Your algorithm has access to an English dictionary, and each time you generate a word using this looping process, you should look it up in the dictionary to make sure that it is actually a legal English word. Another way of visualizing the search for neighboring words is to think of each letter index in the word as being a "spinner" that you can spin up and down to try all values A-Z for that letter. The diagram below tries to depict this:



Another subtle issue is that you **do not reuse words** that have been included in a previous ladder. For example, suppose that you have add the partial ladder cat → cot → cog to the queue. Later on, if your code is processing ladder cat → cot → con, one neighbor of con is cog, so you might want to examine cat → cot → con → cog. But doing so is unnecessary. If there is a word ladder that begins with these four words, then there must be a shorter one that, in effect, cuts out the middleman by eliminating the unnecessary word con. As soon as you've enqueued a ladder ending with a specific word, you've found a minimum-length path from the starting word to the end word in the ladder, so you never have to enqueue that end word again.

To implement this strategy, keep track of the set of words that have already been used in any ladder. Ignore those words if they come up again. Keeping track of what words you've used also eliminates the possibility of getting trapped in an infinite loop by building a circular ladder, such as cat → cot → cog → bog → bag → bat → cat.

It is helpful to test your program on smaller dictionary files first to find bugs or issues related to your dictionary or word searching. We have provided files named smalldict1.txt through smalldict3.txt that you can try. Here is a sample log of execution using a smaller dictionary file:

```
Welcome to CS 106B Word Ladder.
Please give me two English words, and I will change the first into
the second by changing one letter at a time.

Dictionary file name? notfound.txt
Unable to open that file. Try again.
Dictionary file name? oops.txt
Unable to open that file. Try again.
Dictionary file name? smalldict1.txt

Word #1 (or Enter to quit): code
Word #2 (or Enter to quit): data
A ladder from data back to code:
data date cate cade code

Word #1 (or Enter to quit): ghost
Word #2 (or Enter to quit): boo
```

```
The two words must be the same length.

Word #1 (or Enter to quit): marty
Word #2 (or Enter to quit): keith
The two words must be found in the dictionary.

Word #1 (or Enter to quit): kitty
Word #2 (or Enter to quit): kitty
The two words must be different.

Word #1 (or Enter to quit): dog
Word #2 (or Enter to quit): cat
A ladder from cat back to dog:
cat cot cog dog

Word #1 (or Enter to quit):
Have a nice day.
```

PROJECT 2: TWO FLAVORS OF RANDOM SENTENCE GENERATOR [by Cynthia Lee]

***Note:* only Project 1 and Project 2, Part I are required. Project 2 Part 2 is extra credit.**

This project has two parts, each representing one philosophical branch of the field of artificial intelligence, specifically of computational linguistics: (1) Rules-based systems, and (2) Statistical inference systems.

Rules-based systems: In the early days of the field, scholars worked to enumerate all the rules of grammar of human language in big hierarchical list. Part of this effort was the "context-free grammar"[1]. This rule-based approach to artificially-intelligent processing of human language relied heavily on human effort to research, formulate, and organize all the rules.

Statistical inference systems: In the late 1990's, a new movement developed that used statistics to *infer* (guess) the rules based on mounds of evidence, rather than requiring humans to determine and input the rules. Skeptics doubted this random, messy, hands-off approach could ever match the quality of the expertly-curated human approach. But, led by Google, the statistical approach won out, and we've essentially never looked back. Now everything from spam filters to Amazon shopping suggestions, from self-driving cars to your Pandora playlist, from your web ads to your Facebook feed, are generated based on rules and preferences inferred from statistical analysis of mounds of empirical evidence including your previous interactions with the technologies (and the previous interactions of thousands of others).

[1] Fun fact: context-free grammars were invented by Noam Chomsky, now perhaps better known as the politically radical author of *Manufacturing Consent*.

Today, it's hard to come up with examples of the rule-based approach that would be familiar to you, because the statistical approach has so thoroughly dominated. (Aside: this is why you all should be really excited to take CS109! We'll learn the mathematical tools behind these technologies.)

In this assignment, you will compare two simple applications that exemplify the spirit of the two different approaches. Experiment and draw your own conclusions about the relative merits of each!

Project 2, Part I [REQUIRED]: Statistical Random Sentence Generator [by Marty Stepp, edited]

In this part of this assignment, you will write a program that reads an input file and uses it to build a large data structure of word groups called "*N*-grams," as a basis for randomly generating new text that sounds like it came from the same author as that file. You will use the Map and Vector collections from Chapter 5.

Below is an example **log of interaction** between your program and the user:

```
Welcome to CS 106B Random Writer ('N-Grams').
This program makes random text based on a document.
Give me an input file and an 'N' value for groups of words, and I'll create
random text for you.

Input file? hamlet.txt
Value of N? 3

# of random words to generate (0 to quit)? 40
... chapel. Ham. Do not believe his tenders, as you go to this fellow. Whose
grave's this, sirrah? Clown. Mine, sir. [Sings] O, a pit of clay for to the King
that's dead. Mar. Thou art a scholar; speak to it. ...

# of random words to generate (0 to quit)? 20
... a foul disease, To keep itself from noyance; but much more handsome than
fine. One speech in't I chiefly lov'd. ...

# of random words to generate (0 to quit)? 0
Exiting.
```

But what, you may ask, is an *N*-gram?

The "Infinite Monkey Theorem" states that an infinite number of monkeys typing random keys forever would eventually produce the works of William Shakespeare. That's not very useful, due to the amount of time it would take typing individual letters with uniform probability. Instead, we'll choose *words* at random, instead of individual letters? Further, suppose that rather than each word having an equal probability of being chosen, we weighted the probability based on how often that word appeared in Shakespeare's works, and in what context?

Our model will look at *chains of two words* in a row. For example, suppose Shakespeare uses the word "to" 10 times total in a given text, and in 7 of those occurrences it is followed by "be", 1 time by "go", and 2 times by "eat". We can use those ratios when choosing the next word. If the last word we chose is "to", we randomly choose "be" with probability 7/10, "go" with probability

1/10, and "eat" with probability 2/10. **We never choose any other word to follow "to".** We call a chain of two words like this, such as "to be", a *2-gram*.

```
+--------------------+
| Chose "to".        |----> choose "be"    (7/10 chance)
| Next random word?  |----> choose "go"    (1/10 chance)
+--------------------+----> choose "eat"   (2/10 chance)
```

Go, get you have seen, and now he makes as itself? (2-gram)

A sentence of 2-grams isn't great, but look at chains of 3 words (*3-grams*). If we chose the words "to be", what word should follow? If we had a collection of all sequences of 3 words-in-a-row with probabilities, we could make a weighted random choice. If Shakespeare uses "to be" 22 times and follows them with "or" 5 times, "in" 3 times, "with" 10 times, and "alone" 4 times, we could use these weights to randomly choose the next word. So now the algorithm would pick the third word based on the first two, and the fourth based on the (second+third), and so on.

```
+--------------------+----> choose "or"    ( 5/22 chance)
| Chose {"to", "be"}. |----> choose "in"    ( 3/22 chance)
| Next random word?  |----> choose "with"  (10/22 chance)
+--------------------+----> choose "alone" ( 4/22 chance)
```

One woe doth tread upon another's heel, so fast they follow. (3-gram)

You can generalize the idea from 2-grams to *N*-grams for any integer *N*. If you make a collection of all groups of *N-1* words, along with each possible 1 following word, you can use this to select an *N*th word given the preceding *N-1* words. The higher *N* level you use, the more similar the new random text will be to the original data source. Here is a random sentence generated from 5-grams of *Hamlet*, which is starting to sound a lot like the original:

I cannot live to hear the news from England, But I do prophesy th' election lights on Fortinbras. (5-gram)

Each particular piece of text randomly generated in this way is also called a *Markov chain*. Markov chains are very useful in computer science and elsewhere, such as artificial intelligence, machine learning, economics, and statistics.

Algorithm Step 1: Building Map of N-Grams

In this program, you will read the input file one word at a time and build a particular compound collection, a **map** from prefixes to suffixes. If you are building 3-grams (that is, *N*-grams for *N*=3), then your code should examine sequences of 2 words and look at what third word follows those two. For later lookup, your map should be built so that it connects a collection of *N*-1 words with another collection of all possible suffixes; that is, all possible *N*th words that follow the previous *N*-1 words in the original text. For example if you are computing *N*-grams for *N*=3 and the pair of words "to be" is followed by "or" twice and "just" once, your collection should map the key {to, be} to the value {or, just, or}. The table below illustrates the file reading process.

When reading the input file, the idea is to keep a window of *N*-1 words at all times, and as you read each word from the file, discard the first word from your window and append the new word. The following figure shows the file being read and the map being built over time as each of the first few words is read to make 3-grams:

to be or not to be just ... ^	map = {} window = {to, be}
to be **or** not to be just ... ^	map = {**{to, be}** : **{or}**} window = {be, **or**}

to be or **not** to be just ... ^	map = {{to, be} : {or}, **{be, or}** : **{not}**} window = {or, **not**}
to be or not **to** be just ... ^	map = {{to, be} : {or}, {be, or} : {not}, **{or, not}** : **{to}**} window = {not, **to**}
to be or not to **be** just ... ^	map = {{to, be} : {or}, {be, or} : {not}, {or, not} : {to}, **{not, to}** : **{be}**} window = {to, **be**}
to be or not to be **just** ... ^	map = {{to, be} : {or, **just**}, {be, or} : {not}, {or, not} : {to}, {not, to} : {be}} window = {be, **just**}
...	...
to be or not to be just be who you want to be or not okay you want okay	map = {{to, be} : {or, just, or}, {be, or} : {not, not}, {or, not} : {to, okay}, {not, to} : {be}, {be, just} : {be}, {just, be} : {who}, {be, who} : {you}, {who, you} : {want}, {you, want} : {to, okay}, {want, to} : {be}, {not, okay} : {you}, {okay, you} : {want}, **{want, okay}** : **{to}**, **{okay, to}** : **{be}**}
input file, tiny.txt	*resulting map of 3-gram suffixes*

Note that the **order matters**: For example, the prefix {you, are} is different from the prefix {are, you}. Note that the same word can occur multiple times as a suffix, such as "or" occurring twice after the prefix {to, be}.

Also notice that the map **wraps around**. For example, if you are computing 3-grams, perform 2 more iterations to connect the last 2 prefixes in the end of the file to the first 2 words at the start of the file. In our example above, this leads to {want, okay} connecting to "to" and {okay, to} connecting to "be". If we were doing 5-grams, we would perform 4 more iterations and connect to the first 4 words in the file, and so on. This turns out to be very useful to help your algorithm later on in the program.

You **should <u>not</u> change case or strip punctuation** of words as you read them. The casing and punctuation turns out to help the sentences start and end in a more authentic way. Just store the words in the map as you read them.

Algorithm Step 2: Generating Random Text

To generate random text from your map of *N*-grams, first choose a random starting point for the document. To do this, pick a randomly chosen key from your map. Each key is a collection of *N*-1 words. Those *N*-1 words will form the start of your random text. This collection of *N*-1 words will be your **sliding "window"** as you create your text.

For all subsequent words, use your map to look up all possible next words that can follow the current *N*-1 words, and randomly choose one with appropriate weighted probability. If you have built your map the way we described, as a map from {prefix} → {suffixes}, this simply amounts to choosing one of the possible suffix words at random. Once you have chose your random suffix word, slide your current "window" of *N*-1 words by discarding the first word in the window and appending the new suffix. The following diagram illustrates the text generation algorithm.

Action(s)	Current (*N*-1) "window"	Output so far
choose a random start	`{"who", "you"}`	`who you`
choose new word; shift	`{"you", "want"}`	`who you want`
choose new word; shift	`{"want", "okay"}`	`who you want okay`
choose new word; shift	`{"okay", "to"}`	`who you want okay to`
...	`...`	`...`

Note that in our random example, at one point our window was {want, okay}. This was the end of the original input file. Nothing actually follows that prefix, which is why it was important that we made our map **wrap around** from the end of the file to the start, so that if our window ever ends up at the last *N*-1 words from the document, we won't get stuck unable to generate further random text.

Since your random text likely won't happen to start and end at the beginning/end of a sentence, just prefix and suffix your random text with "..." to indicate this. Here is another partial log of execution:

```
Input file? tiny.txt
Value of N? 3
# of random words to generate (0 to quit)? 16
... who you want okay to be who you want to be or not to be or ...
```

Your code should check for several kinds of **user input errors**, and not assume that the user will type valid input. Specifically, re-prompt the user if they type the name of a file that does not exist. Also re-prompt the user if they type a value for *N* that not an integer, or is an integer less than 2 (we are only interested in 2-grams and longer). You may assume that the value the user types for *N* is not greater than the number of words found in the file. See the logs of execution on the course web site for examples of proper program output for such cases.

Get Creative

Along with your program, submit a file **myinput.txt** that contains a text file that can be used as input for Part B. This can be anything you want, as long as it is non-empty and is something you gathered yourself (not just a copy of an existing input file). This is meant to be just-for-fun; for example, if you like a particular band, you could paste several of their songs into a text file, which leads to funny new songs when you run your N-grams program on this data. Or gather the text of a book you really like, or poems, or anything you want. This is worth a small part of your grade on the assignment.

Development Strategy and Hints for N-Grams

This program can be tricky if you don't develop and debug it step-by-step. Don't try to write everything all at once. Make sure to **test** each part of the algorithm before you move on. See the

Homework FAQ for more tips.

- Think about exactly what **types of collections** to use for each part. Are duplicates allowed? Does order matter? Do you need random access? Where will you add/remove elements? Etc. Note that some parts of each program require you to make compound collections, that is, a collection of collections.
- Test each function with a very **small input** first. For example, use input file tiny.txt with a small number of words so you can **print your entire map** and examine its contents.
- Recall that you can **print** the contents of any collection to `cout` and examine its contents for debugging.
- Remember that when you assign one collection to another using the = operator, it makes a full copy of the entire contents of the collection. This could be useful if you want to copy a collection.
- To choose a random prefix from a map, consider using the map's keys member function, which returns a Vector containing all of the keys in the map. For **randomness** in general, include "random.h" and call the global function `randomInteger(`***`min, max`***`)`.
- You can loop over the elements of a vector or set using a for-each loop. A for-each also works on a map, iterating over the keys in the map. You can look up each associated value based on the key in the loop.
- Don't forget to test your input on **unusual inputs**, like large and small values of *N*, large/small # of words to generate, large and small input files, and so on. It's hard to verify random input, but you can look in smallish input files to verify that a given word really does follow a given prefix from your map.
- Your solution should match the flow and prompts shown above and in the sample outputs on the assignments web page.

Project 2, Part 2 [EXTRA CREDIT]: Grammar Rules Random Sentence Generator [by Julie Zelenski]

Over the past two or three decades, computers have revolutionized student life. In addition to providing entertainment and distraction, computers also have also facilitated all sorts of student work. 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. Neglected, that is, until now.

The Random Sentence Generator is a marvelous piece of technology that creates random sentences from a structure known as a **context-free grammar**. A grammar is a construct describing 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:

- Tactic #1: Wear down the TA's patience.

I need an extension because I had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the four-square semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.

- 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 just didn't feel like working.

What is a grammar?

A grammar is a set of rules for some language, be it English, Java, C++, or something you just invent for fun. ☺ If you continue 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 CFG for generating poems:

```
<start>
1
The <object> <verb> tonight.

<object>
3
waves
big yellow flowers
slugs

<verb>
3
sigh <adverb>
portend like <object>
die <adverb>

<adverb>
2
warily
grumpily
```

According to this grammar, two syntactically valid poems are **"The big yellow flowers sigh warily tonight."** and **"The slugs portend like waves tonight."** Essentially, the strings in brackets (<>) are variables that expand according to the rules in the grammar.

More precisely, each string in brackets is known as a **nonterminal**. A nonterminal 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's something of a dead end—that no further expansion is possible.

A **definition** consists of a nonterminal and a list of possible **productions** (or **expansions**). There will always be at least one and potentially several productions for each nonterminal. A production is just a text string of words, some of which themselves may be non-terminals. A production can be the empty string, which makes it possible for a nonterminal to evaporate into nothingness. An entire definition is summarized within a grammar text file as:

```
<verb>                    ⇐ the first line names the nonterminal and is delimited by < and >
3                         ⇐ the second line is always the number of possible expansions
sigh <adverb>             ⇐ the third line is the first possible expansion
portend like <object>     ⇐      followed by another expansion if there is a second one
die <adverb>              ⇐          followed by another expansion if there is a third one, etc
                          ⇐ for readability, there's a blank line after each definition, including the last one
```

You always begin random sentence generation with the single non-terminal **<start>** as the working string, and iteratively search for the first nonterminal[2] and replace it with any one of its possible expansions (which may and often will include its own nonterminals). Repeat the process over and over until all nonterminals are gone.

```
<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, a second generation would almost certainly produce a different sentence.

Your program should repeatedly prompt the user for a grammar file (understood to be the **grammars** subdirectory), read in the grammar, and generate three random sentences separated by a blank line. Only when the user hits return without actually typing in anything should you end the program. Using the sample application as a guide, you are to make all design and implementation decisions.

Some simplifying assumptions:

- You may assume that the grammar files are properly formatted, and that the grammars themselves are well formed. All grammars will include a **"<start>"** nonterminal, and

[2] This problem could also be solved using recursion, but we ask that you don't solve this recursively, but instead solve it using iteration.

all nonterminals will expand to one or more definitions (which themselves may and often will include other nonterminals).

- You needn't worry about word wrap as you generate and print out über-long sentences.

GRADING AND GENERAL REMARKS (this applies to all parts)

- All items mentioned in the "Grading and General Remarks" section of the previous assignment spec also applies here. Please refer to that document as needed. Note the instructions in the previous assignment about style, pass by reference, passing by `const` reference, and so on.
- Don't forget to cite any sources you used in your comments. Please read the entire CS106 Honor Code on the course website.
- Refer to the course Style Guide for a more thorough discussion of good coding style.
- *Algorithms:* You should follow the general algorithms as described in this document and should not substitute a very different algorithm. In particular, you should not write a recursive algorithm for finding word ladders or N-grams.
- *Collections:* Additionally, on this assignment part of your Style grade comes from making intelligent decisions about what kind of collections from the Stanford C++ library to use at each step of your algorithm, as well as using those collections elegantly. As much as possible, pass collections by reference (and const reference, where possible), because passing them by value makes an expensive copy of the collection.
- Do not use pointers, arrays, or STL containers on this program. You should also avoid expensive operations that would cause you to reconstruct bulky collections multiple times unnecessarily. For example, in N-grams, generate the map of prefixes exactly once; do not regenerate it each time the user asks to generate random text.