CS 3520—Lua Markov chain babbler

Lua is a popular language, and there are many resources available online to help you learn it. Please find whatever tutorials and references you need. You may also wish to investigate videos that introduce the language. You will probably need to do more than rely on lectures to complete this assignment.

Markov chains

A Markov chain is a mathematical system that transitions from one state to the next by obeying probabilities, but without any memory. For example, suppose you have three states A, B, and C, and the probability of moving from one to the next (A to B, B to C, or C to A) is 50%, the probably of moving backwards (B to A, C to B, A to C) is 25%, and the probability of remaining in the same state is 25%. The chain of states that would result is a Markov chain.

If you started in A, you could randomly choose the next state, with a 50% chance of moving forward to B, and 25% probabilities of staying at A or moving to C instead. Let's say you transitioned to B. Then there would be a 50% chance of moving to C, 25% chance of remaining at B, and 25% chance of moving back to A. The history of the system (the fact that it was in state A before moving to B) does not impact the system.

In a Markov chain babbler, we load a corpus of text, parse it into words, and then create a new sequence of words based on the original text. The new output mimics the input somewhat, but uses a Markov chain to put words together in a new way.

Your babbler will be configurable with a single parameter called *n*. The output will have the property that every group of *n* words in the output (called an *n-gram*) must have appeared in the input text somewhere. *n*-grams are also called shingles. For our examples, we will assume *n*=3, but your code should work equally well with other values of *n*.

Parsing

This project will use a mix of C and Lua. Here is how that mix breaks down:

- The C program starts out in `main`
- It checks and parses the command-line arguments.
- It initializes the Lua environment and registers C functions that can be called from Lua
- It calls a Lua function to drive the rest of the process
- The Lua code calls a C function to load the input file, returning it as Lua string
- The Lua code calls a C function to parse the input string. It passes in the string and an offset (see below) and gets back a single new token and a new offset. It calls this many times, resulting in a list of tokens in Lua.
- The Lua code generates n-grams, then shingles, then a table of prefixes mapped to last words (see below).
- The Lua code babbles the requested number of words to the console

To start out, you will need to parse an input file. You can get input text from anywhere (try Project Gutenberg), but the results are best when the test has a distinctive style and subject matter. Our parsing rules will be simple:

- Every sequence of one or more non-whitespace characters in the input is called a *raw token*.

- Every raw token with all non-alphanumeric characters removed and with all letters converted to lower case is called a *token*.

Your first task is to parse a file of input into a list of tokens, keeping only those tokens that are non-empty. If a token has zero characters, it should be discarded (this occurs when you find a sequence of punctuation characters with no letters or numbers in it).

You must open and read the file in a C function, which should be called from Lua. It should take the name of the file as a parameter, and return the entire contents of the file as a Lua string.

Next, you should parse the string into a series of tokens. You should write a C function that takes the file content string and an offset into it file as parameters. It returns the next available token as a Lua string, and the offset into the string where the search for the next token should begin. If there are no tokens left in the string, it should return the empty string instead of a valid token. The function should use the following rules:

- Skip all whitespace characters
- Mark the beginning of a raw token

- Scan until the next whitespace character
- Note where the end of the raw token is
- Copy the alphanumeric characters of the raw token into a new string, converting everything to lower case as you go
- Return the new token, and the offset of the next character past the end of the raw token

You should be able to call this function repeatedly from Lua, giving it the string and the offset into the string each time. Note that the C function should not store any state: it should get all the information it needs from the calling parameters. On the Lua side, you should track the list of tokens, and the offset of the next place to search for a token.

Using these functions (and any others you find helpful), write a Lua function that parses an input file into a list of tokens.

Forming n-grams

With a list of tokens in hand, it is time to create all possible *n*-grams possible in a process called *shingling* (named after the way shingles on a roof overlap each other). Assume for a moment that *n* is 3. If your input tokens were

```
three blind mice lost their tails
```

You would produce the following *n*-grams:

1. three blind mice
2. blind mice lost
3. mice lost their
4. lost their tails
5. their tails three
6. tails three blind

Notice that the *n*-grams at the end of the list wrapped around back to the beginning of the list, resulting in exactly one *n*-gram for each token of the input. Duplicate *n*-grams are okay and should not be filtered out.

Making sure you can form the correct *n*-grams, and make sure your code works correctly for different values of *n*.

Building the lookup table

Use a table to store the *n*-grams. We need the ability to find every *n*-gram with a given *n-1* length prefix. For our example above, this means being able to easily find “blind mice lost” given “blind mice”.

The key for each *n*-gram is a string with the first *n-1* tokens in the *n*-gram, and the value to store is a list of the final tokens of each *n*-gram. For example, if you had the following *3-grams*:

- eat the apple
- eat the pear
- eat the pickle
- eat the apple

You would produce an entry in the map with key “eat the” and value `{"apple","pear","pickle","apple"}`. Note again that duplication is okay as long as it faithfully reflects the contents of the input text.

User interface

Your executable should accept two required parameters (the name of the input text file and the number of words to produce) and one optional parameter (the value of *n*). If *n* is omitted, use a default value of 3. For example:

```
./babbler drseuss.txt 100 4 # babble 100 words using 4-grams
./babbler drseuss.txt 50    # babble 50 words using 3-grams
```

Babbling

Now it is time to randomly generate a stream of words. Pick a random *n*-gram from the input. Do not grab one from the map, as it is difficult to do so in a truly random fashion. Instead, grab one directly from the input token list, or if you made a list of *n*-grams, pick one at random from the list.

Output only the last word from the *n*-gram. Then discard the first token from the *n*-gram, giving you a prefix of length *n-1*. Use this to lookup all words that followed this prefix in the original text, and randomly pick one of those words. Use it to form the next *n*-gram.

For example, if you had “three blind mice” as your *n*-gram (and you output “mice” to the console), you would discard “three” and look up all *n*-grams that start with “blind mice”. Say the list of 3rd words included “lost”, “ate”, and “ran”. You should randomly pick one of those 3rd words, say “ate”, and append it onto “blind mice”, yielding “blind mice ate”.

Then output the new last word (“ate” in our example”) and repeat by chopping off the first word (“blind”), finding all *n*-grams that start with the *n-1* length prefix (“mice ate” in our example), and randomly choose the next word from that list.

Repeat this process until you have emitted exactly the number of tokens that the user asked for when invoking the babbler.

Markov chains revisited

Let us examine why the output is a Markov chain.

The states are *n*-grams from the original text. The next token (and thus the next *n*-gram) chosen at each step is picked from the list of alternatives generated from the original text. By retaining duplicates, the probability that a given word will be chosen is driven by the frequency with which it occurred in the original text.

The tokens are printed to the console as you go, but the history of the babbled sequence is ignored when choosing the next word to babble.