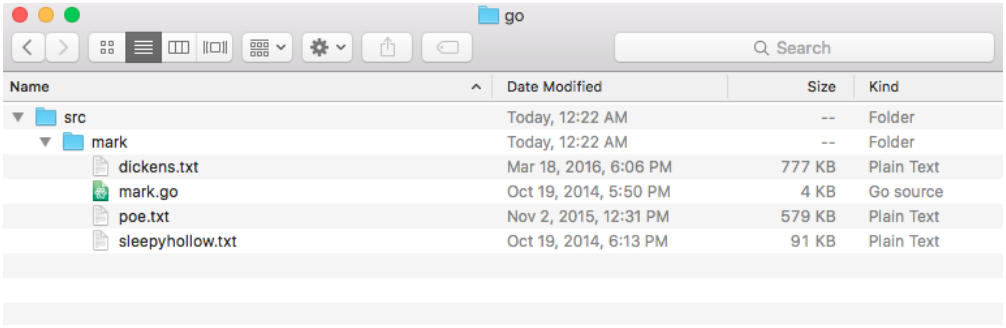
Homework: Mark V. Shaney
Programming for Scientists

1. Set up

Download the template from the course website, and unzip it into the `src` directory inside of your `go` directory. You should now have directories that look like this:



2. Writing data to files

2.1 Opening files for writing

When you use the `os.Open` function, it opens an existing file for reading data. If you want to create a new file to write data to instead, you use the `os.Create` function:

```
outFile, err := os.Create("model.txt")
if err != nil {
    fmt.Println("Sorry: couldn't create the file!")
}
```

2.2 Writing to files

The screen you have been writing to using the `fmt.Print` statements all along is in fact a file, opened automatically for writing, stored in the file variable `os.Stdout`. The name `Stdout` is short for "standard output" but a better name might be "default output" — this is where output goes unless you specify someplace else. (`os.Stdin` is the default input file that you can read from to read what the user is typing.) There are variants of `fmt.Print` that start with `F` that let you specify a different file. Example, the following two print statements are exactly equivalent:

```
var x int = 10
fmt.Println("hi", "there", x)
fmt.Fprintln(os.Stdout, "hi", "there", x)
```

If you want to write to a file you have created with `os.Create` you can simply use the variable returned by `os.Create` in place of `os.Stdout`:

```
var x int = 10
outFile, _ := os.Create("myNewFile.txt")   // open the file, ignore any error!
fmt.Fprintln(outFile, "hi", "there", x)    // write hi there 10 to the file
```

There are 3 functions like `Print` to output in various formats to files:

- `fmt.Fprintln(FILE, item1, item2, ...)` — Write values of the items to `FILE`, separated by spaces, and then start a new line.
- `fmt.Fprint(FILE, item1, item2, ...)` — Same as `fmt.Fprintln` except do not start a new line after writing the data.
- `fmt.Fprintf(FILE, Format, item1, item2, ...)` — Write items to `FILE`, using the `Format` (see below). This is the most powerful, general printing function.

The `Format` parameter of `fmt.Fprintf` is a string that specifies how to write each of the items and what to write between them. For example:

```
var m,d,y int = 10, 31, 2016
var name string = "Halloween"
fmt.Fprintf(outFile, "%v is on %v/%v/%v\n", name, m, d, y)
```

will print

```
Halloween is on 10/31/2016
```

Each `%v` in the format string is replaced by the data in the corresponding variable. Special sequences in the format starting with a `\` will print special characters, the most important of which is `\n` which means start a new line. To print a `%` or `\` character directly, you have to list them twice:

```
fmt.Fprintf(outFile, "Name=%v\nDay=%v\nMonth=%v\nYear=%v\n%% %% %%\n", name, d, m, y)
```

will print:

```
Name=Halloween
Day=31
Month=10
Year=2016
% % %
```

The format string is even more powerful: you can use other codes starting with `%` to control how the data is formatted. For example, using `%.2f` will print a floating point number printing out only 2 digits after the `"."`. You can read about these variants here https://golang.org/pkg/fmt/.

2.3 Closing files

When writing, it's essential that you close the file you are writing to (`os.Stdout` is closed automatically when your program ends). This is because it is not required that the system actually write your data to the disk until you close the file. This delay is because writing to disk can be slow, and Go is allowed to wait until it has enough data to write out to make actually accessing the disk worth the time do to so. Closing a file you have created is easy: you call the `Close()` method on the file variable:

```
outFile.Close()
```

Defer statements. Since closing files is so important, Go provides a neat way to ensure that you don't forget to do it. The `defer` statement lets you specify a function to call whenever the current function ends. The most common use of this is to ensure files are closed:

```
func saveStringsUpTo(filename string, s []string, stop string) {
    out, err := os.Create(filename)
    if err != nil {
        fmt.Println("Error! couldn't create", filename)
    }
    defer out.Close()
    for x := range s {
        if x == stop {
            return // out.Close() is automatically called here
        }
        fmt.Fprintln(x)
    }
    // out.Close() is automatically called here
}
```

`defer` is nice for two reasons: (1) You can put the `Close` call right after you are sure you opened the file so that the open and close instructions appear near each other in the code, and (2) No matter how your function ends, `Close` will be called (even if there are return statements in the middle of your function).

3. Assignment

3.1 Simulating an English speaker

In English, we have a good sense of which words are going to follow a given set of words. For example, there is a good chance that the phrase `No matter` will be followed by the word "how" or "what". In the mid-1980s, Rob Pike (Co-inventor of Go) and colleagues used this property of human language to write a program to create fake-but-believable posts on internet forums. They used a large amount of English text as training data; for every pair of words they observed, they computed the frequency with which they saw the third word following them. For example, given the text:

> no matter how hard you try no matter can escape a black hole

They would compute the following table of frequencies:

"" ""	no=1
"" no	matter=1
no matter	how=1; can=1
matter how	hard=1
how hard	you=1
hard you	try=1
you try	no=1
try no	matter=1
matter can	escape=1
can escape	a=1
escape a	black=1
a black	hole=1

The special empty string entries `""` indicate places where there was no previous word (i.e. the start

of the input). In most cases in this example, there is only one choice for the next word, but for the pair `no matter`, there are two choices, each observed once. This kind of table might be stored in a variable of type:

```
map[[2]string]map[string]int
```

That is: a map from pairs of strings to tables that give the frequencies of each word that followed that pair. (Note: you don't have to implement it in this way.)

Once you have created such a table, you can *generate* English-like text in the following way:

1. Choose a random pair of words to start with, say `x y`
2. Repeatedly choose the next word to output by looking at the table for the two previous output words and picking a word in proportion to how frequently you observed that word in that context.

This process is called a Markov chain of order 2: its *order* is two because you look at the previous 2 words to decide on the next word. It is "Markov" because this is the only information influencing your decision.

Markov chains can model a large range of systems: the performance of a stock on day i might be modeled based on its performance on days $i-1$ and $i-2$. Because DNA is a language of its own, a random DNA sequence generator might be modeled by outputting `A`, `C`, `G`, or `T` based on the previous 2 bases output, and so on. There's nothing special about 2: using a larger order makes the output more realistic, but it requires more input data to train the frequencies.

3.2 Mark V. Shaney

Pike and colleagues put this idea to a fun use: posting to (some would now say "trolling") internet discussion forums. A famously typical post begins:

> It looks like Reagan is going to say? Ummm... Oh yes, I was looking for. I'm so glad I remembered it. Yeah, what I have wondered if I had committed a crime. Don't eat with your assessment of Reagon and Mondale. Up your nose with a guy from a firm that specifically researches the teen-age market. As a friend of mine would say, "It really doesn't matter"... It looks like Reagan is holding back the arms of the American eating public have changed dramatically, and it got pretty boring after about 300 games.

This seems like "plausible" English, even though of course what it means is nonsense.

You can see a simplified version of their program, re-written in Go, here:

- https://golang.org/doc/codewalk/markov/

and read about its reception here:

- http://web.archive.org/web/19961119143254/http://www.sincity.com/penn-n-teller/pcc/shaney.html
- http://www.nature.com/scientificamerican/journal/v260/n6/pdf/scientificamerican0689-122.pdf

We are still far from having effective language simulators. For example, in 2016, Microsoft created a chatbot called "Tay Tweets" that had to be taken down after it was manipulated to tweet offensive

remarks:

https://www.theguardian.com/technology/2016/mar/24/tay-microsofts-ai-chatbot-gets-a-crash-course-in-racism-from-twitter

3.3 What you will do

Your job in this assignment is to take the initial implementation of Mark V. Shaney (linked above and included in the template) and extend it to (a) save a frequency table to a file, (b) read a frequency table from a file, and (c) actually use the frequencies (the simple version assumes every word occurs at the same frequency).

(Note: all subsequent commands will assume you are using a Mac and use `./mark` instead of `mark.exe` as the first word of the command to run the program.)

Your program should be runnable with the command (Mac):

```
./mark COMMAND options
```

where COMMAND is either `read` or `generate`.

If COMMAND is `read` then the user should run your command like this:

```
./mark read N outfilename infile1 infile2 ....
```

where `outfilename` gives the file to save the table to, and `infile1` and so on give the files to read, one after another. The user can specify any number of input files. N specifies the number of words to use for context (aka the order of the chain which is 2 in the example above, but your program should support contexts of length ≥ 1). Your program should read each input file, create the frequency table, and then save that frequency table to the file `outfilename`. To make your work easier to interpret, you may want to *sort* the frequency table when writing it to file, so that the lines are printed in lexicographic (i.e., dictionary) order.

If COMMAND is `generate` then the user should run your command like this:

```
./mark generate modelfile n
```

where `modelfile` is the name of a file saved using the `read` command and n is the number of words to output. Your program should read the frequency table in the file `modelfile` and then use it to generate n words of output.

It may be the case that you get stuck before you reach n words; if that happens, just stop.

3.4 Format of the model frequency file

The first line of the file should contain a single number which is the order of the chain (2 in the examples above). Each subsequent line should save the frequencies in the format:

```
contextWord1 contextWord2 ... nextWord1 count2 nextWord2 count2 nextWord3 count3 ...
```

In the example above, the file would be:

```
2
"" "" no 1
"" no matter 1
no matter how 1 can 1
```

```
matter how hard 1
how hard you 1
hard you try 1
you try no 1
try no matter 1
matter can escape 1
can escape a 1
escape a black 1
a black hole 1
```

Use the empty string "word" `""` to indicate an empty word (at the start of an input file).

3.5 Tips on how to start

First, install the template linked from Diderot and make sure you can:

```
go build
./mark < poe.txt
```

(On Windows you may have to type `.\mark`.) If it works, this should print out something that looks like Poe's writing. The syntax `< filename` means treat `filename` as the standard input as if the user was typing it. Play around with this a bit, perhaps feeding your own text files. Or even trying `./mark < mark.go` for a self-referential meta experience. This ensures you are set up to begin coding. **Do this today.**

Next, read through the code in `mark.go` as described here:

- https://golang.org/doc/codewalk/markov/

and understand how the program works.

Next, modify the program to record the frequencies along with the words in some data structure.

Next, modify the output generation function `Generate` to use the frequencies that you've stored.

Next, modify the command line parsing to match what is described in this assignment.

Finally, write the code to read and write the frequency table.

3.6 Some notes

1. For our purposes, words are sequences of space-separated characters, where case matters. In other words, `great`, `great!` and `Great` are all different words.

2. There is one strangeness about Go and maps of structs that we have already encountered and that becomes relevant here; you can't change the value of a field in a struct that is in a map:

```
type S struct {
   name string
   id int
}
var M map[string]S = make(map[string]S)
M["Dave"].id = 10  // ERROR!
```

Hopefully this behavior will be fixed in a future release of Go. But the solution is simple: make a copy of the struct, modify the copy, and then assign the whole struct to the map entry:

```
x := M["Dave"]
x.id = 10
M["Dave"] = x
```

4. Learning outcomes

After completing this homework, you should have

- gotten experience reading and modifying existing code
- practiced using methods
- practiced using structs
- gained an understanding of Markov chains
- gotten additional practice reading from files
- learned how to write files