The Go Programming Labs

For each module in the course, there is a corresponding lab. Each lab has a number of problems related to the material covered in the module. The labs have been designed with the following in mind:

1. They are not all intended to be challenging programming problems, and some of the problems may even seem a bit simplistic in a few cases in order to illustrate a concept or idea.
2. One primary goal of the labs is to help you clarify and understand the concepts about Go presented in the lectures Sometimes we do not know how much we have learned until we actually try to do something with that knowledge or have to explain what we are doing.
3. You should be continuously self-evaluating your your understanding of the material by asking yourself if you understand what is happening in code and why it is happening that way. This is the most critical learning point of the labs. If you are not understanding what is happening in the lab, use the classroom resources to sort it out – ask the instructor or other students.
4. As the labs progress, there will be more programming problems and less conceptual questions. Some of programming problems in later modules can be quite challenging. A number of them are based on standard problems used in teaching Go.

Discussion Questions

Most of the labs contain discussion questions. A discussion question usually asks you to try something and then explain the result which may include being asked to speculate on the reasons why you think the Go code behaved the way it did.

The purpose of these questions is to get past the "I wrote the code, ran it and got the right answer but I have no idea what I did" approach to doing labs. You are experienced technical people and the labs are your opportunity to increase your understanding of Go, not just to do some rote exercises.

At the end of each lab, the instructor will solicit answers to the discussion questions from the class so be prepared with an answer if you have one. If you do not have a solution to a question, it may indicate something that you missed something in the presentation or misunderstood it and should revisit the topic. The ability to answer the questions are your own mini self-assessment tool.

Generally the answers to the discussion questions are points that have been presented in the lectures or can be easily deduced from the lecture content and the material in the student manual. It's also a good idea to get used to looking up answers in the on line Go resources.

For some of the discussion questions, especially the ones that start with “How would you..” there may be more than answer to the question. In this case, try to evaluate what you think the best answer is and why or at least what the benefits and costs are to each alternative answer.

Lab 1: Getting Started with Go

The purpose of this initial lab is get you started in the Go programming environment and up to speed on the basics of writing and executing Go programs either using an IDE or at the command line. It is recommended that if you are using an IDE, that you also try using the command line as well.

Specific Objectives:

1. To validate your Go installation
2. To experiment using the Go tool to compile a Go program
3. To experiment with basic Go syntax
4. To experiment with the Go formatter

Part 1: Verifying your Go Installation

1. If you are working on a ProTech Virtual machine, you can skip this step,
2. At a command prompt, type the command **`go`**
3. You should see the output on the next page.
4. If you do not see this output, then check to make sure that Go is installed correctly on your machine. Review the Go setup steps in Module One of the Student Manual.

Part 2: Using `go run`

1. In a text editor create a file called **`hello.go`**
2. Create a working directory somewhere to save the file to. For the purposes of this lab, we will assume you are working in windows and have are using a directory called C:\gostuff
3. Save the file as **`C:\gostuff\hello.go`**
4. Enter the code in the file to impement the "Hello World!" program from the lecture.
5. It is strongly suggested that you try to type the code in first without looking at the lecture slides, and then compare it to the code in the slide.
6. Do not cut and paste the code since you will get a better assessment onfyour understanding if you actually write the code, even if you are just transcribing what you see.
7. At the command prompt, locate to the directory containing your **`hello.go`** file.

```
Usage:

        go command [arguments]

The commands are:

        build       compile packages and dependencies
        clean       remove object files
        doc         show documentation for package or symbol
        env         print Go environment information
        fix         run go tool fix on packages
        fmt         run gofmt on package sources
        generate    generate Go files by processing source
        get         download and install packages and
                    dependencies
        install     compile and install packages and
                    dependencies
        list        list packages
        run         compile and run Go program
        test        test packages
        tool        run specified go tool
        version     print Go version
        vet         run go tool vet on packages
```

8. Use the following command to compile and run your file.

   `go run hello.go`

9. You should see the output from your application.
10. Verify that the the directory **`C:\gostuff`** is unchanged, ie. there is no executable in the directory

Discussion Question 1-1

1. Try running the command in step 8 above without the file name (ie. **`go run`**) and notice the output.
2. Create a copy of **`hello.go`** and call it **`bye.go`**. Change the output string to "Goodbye Cruel World"

3. Run the following commands and examine the outputs.

   ```
   go run bye.go
   ```

   ```
   go run hello.go
   ```

4. Explain why you got those outputs.
5. Now run the command

   ```
   go run *.go
   ```

6. Explain the result of this command.

Part 2: Using go build

1. For this part, we will use the same hello.go file as in the previous part.
2. Delete the bye.go file from the previous part.
3. At the command prompt, execute the command

   ```
   go build hello.go
   ```

4. Verify there is an executable in the directory now with the name hello or hello.exe (depending on your operating system).

Discussion Question 1-2

1. While in the same directory that `hello.go` is in, run the following command

   ```
   go build
   ```

2. Examine the contents of the directory

   Explain what happened. Can you offer an explanation of why Go does this?

Part 3: Syntax Experimenting

While Go has a C-style syntax, there are some things that we can't do in Go syntactically that we can in other C-style languages.

1. In the lecture there was an example of importing both the `fmt` and `strings` packages.
2. In your `hello.go` program, import both packages but do not change anything else (ie. don't use the `strings.ToUpper()` function).
3. Try to compile the program and note the compile error.

4. Remove the **`strings`** package import.
5. Put a newline character just before the opening brace "{" for the main function, ie. the "{" in now on the next line like this:

```
func main()
{
```

6. Compile and notice the error.

Discussion Question 1-3

It almost seems overly picky to enforce this sytnax rule. Why do you think Go does this?

Part 4: Go Formatting

1. Create a new source file called **`junk.go`**. It can be in the same directory you have been working in so far.
2. Add the following one line exactly as it is here to the file

```
package main;import "fmt";func main() {fmt.Println("Hello World!");}
```

3. Use go run junk.go to verify that this file does compile.
4. Now run the following command:

```
gofmt junk.go
```

5. Notice that the output of this command is a reformmated program listing.
6. You can overwrite the original file with a formatted version by running

```
gofmt -w junk.go
```

7. Use the -w option to verify that the file is in fact overwritten

Discussion Question 1-4

Why have a standard formatting and a formatting tool? What would be the advantage to this over the free form formatting of Java and C?

1. Not all of the formatting was reworked. Try starting with these two lines following in the **`junk.go`** file instead of the single line you used before:

```
package main;import "fmt";func main() {
fmt.Println("Hello World!");}
```

2. Now rerun the **`gofmt`** command on it and notice the formatted output this time is different.

Can you explain the difference in the two outputs?

Lab 2: Variables and Data Types

This lab is intended to allow you to explore some of the behavior of variables and data types. There are no challenging programming problems in this lab since it is more of a chance to poke Go and see how it works and an opportunity to try to break things when it comes to working with Go variables. The lab has a secondary objective of just getting you more familiar with working with Go syntax and Go coding style in general.

Part One: The int data type

The int data type is either 32 bits or 64 bits wide. This means that it is identical to either an int32 or int64.

Discussion Question 2-1

Why does the following code generate an error when we try to assign an int to an int32 or an int 64?

```
// Lab 02-01  That pesky "int" type

package main

func main() {
  var i int
  var i32 int32
  var i64 int64
  i = 0
  i32 = i
  i64 = i
}
```

```
[lab02]$ go run lab02-01.go
./lab02-01.go:10: cannot use i (type int) as type int32 in assignment
./lab02-01.go:11: cannot use i (type int) as type int64 in assignment
```

Part 2: Initializing Variables

These next two problems are more or less just experimenting. You may want to play with some of the sample code that listed here. The source code is available to you but if you are new to Go, you probably should type it in yourself to get the feel for writing code in Go.

The code in the following example is similar to what we saw in class – it is a list of initializers.

```
 // Lab 02-02  List of intitalizers

package main

import "fmt"

var i, b, str, x = 89, true, "hi", float32(45)

func main() {
  fmt.Printf("i=%T b=%T str=%T x=%T \n", i, b, str, x)
  fmt.Printf("i=%d b=%t str=%s x=%f \n", i, b, str, x)
}
```

```
[lab02]$ go run lab02-02.go
i=int b=bool str=string x=float32
i=89 b=true str=hi x=45.000000
```

Discussion Question 2-2

In other programming languages, if there are more literals than variables, the extra literals are just ignored; or if there are more variables than literals, then the extra variables are unitialized or set to some default zero value. Why do you think Go gives an error in either of these cases instead of either one of the options described here?

Part 2: Variable Scoping

This lab requires you to modify and run the code in the file lab02-03.go

The following listing shows the same sort of variable shadowing that we saw in the lecture. The variable "i" is declared at the package level. The function i_print() just is useful in accessing the correct package version of "i" even after it has been shadowed. If this is not clear to you then look at the code and figure out what the scope of the shadowing copy of "i" is and whether or not the function is in that scope.

In order to see if the various copies of "i" are the same or different, the code prints out the address of the variable rather than its value. This was covered in the material when we talked about pointers.

```
// Lab 02-03  Variable Scoping
package main

import "fmt"

var i = 1

func i_print() {
  fmt.Println("pkg level &i = ", &i)
}

func main() {
  i_print()
  i := 2
  fmt.Println("first assigment &i=", &i)
  i, j := 3, 1
  j += 1 // just so the compiler doesn't complain
  fmt.Println("second assigment &i=", &i)
  i_print()
}
```

1. Either type in the code or use the file provided and run the code.
2. The variable "i" appears on the LHS of an assignment statement at three places in the code.
3. By comparing the addresses of “i” at different places, you should be able to see the shadowing clearly.
4. Change the code as shown in the line below. Now re-run the code and examine how the output has changed from the previous run.
5. Can you explain the change in output with respect to the address of i?

```
// Lab 02-03  Variable Scoping (modified)
...
func main() {
  i_print()
  i = 2   <-- changed from := to =
  fmt.Println("first assigment &i=", &i)
  i, j := 3, 1
  j += 1 // just so the compiler doesn't complain
  fmt.Println("second assigment &i=", &i)
  i_print()
}
```

Part 3: Fun with Strings

1. The following code is available in the lab section in the file lab02-04.go. You can't actually type the listing in because of the characters in the strings. In fact what you see below has a graphic pasted over it to show you what the listing should look like.
2. Notice that each string is two characters long.
3. Run the code and explain the results, specifically, why is the length of the string not always 2?

```
// Lab 02-04  Strings
package main

import "fmt"

func main() {
  Cyrillic := "中 Я"
  Latin := "a b"
  Hanzi := "亲 仇"
  fmt.Println("Length of", Latin, " is", len(Latin))
  fmt.Println("Length of", Hanzi, " is", len(Hanzi))
  fmt.Println("Length of", Cyrillic, " is", len(Cyrillic))
}
```

Part 4: Constants

1. Consider the code on the next page.
2. You should run it yourself to verify the results.
3. Change the following line

   ```
   const x = 4
   ```

   to

   ```
   var x = 4
   ```
4. Verify that this is a compile error. Why does the change cause the program to longer compile?

```
// Lab 02-05  Constants
package main

import "fmt"

const x = 4
const y = 8 / x

func main() {
  fmt.Println("y =", y)
}
```

Part 5: iota

This is a programming problem so you will have to write the code yourself from scratch. The complete solution in the file lab02-06-solution.go

1. Using iota, write two sets of constants
2. The first set should be named two_sq, three_sq, four_sq and the the square of 2, 3, 4 (ie. their values should be 4, 6 and 9)
3. The second set should be named two_cu, three_cu, four_cu and the the cubes of 2, 3, 4 (ie. their values should be 8, 27 and 64)

Part 6: Pointers

1. Add one line to the program listing on the next page to produce the output shown.
2. The solution is in lab02-07-solution.go

```
// Lab 02-07  Pointers
package main

import "fmt"

var i, j = 1, 2
var pi, pj = &i, &j

func list() {
  fmt.Println("i=", i, "j=", j)
}

func main() {
  list()
  i, j := 100, 101
  i, j = j, i
  // insert line here
  list()
}
```

```
[lab02]$ go run lab02-07.go
i= 1 j= 2
i= 2 j= 1
```

Lab 3: Operators and Control Structures

The purpose of this lab is to provide you with a chance to experiment with some of the topics covered in module 3. The exercises are simple but do require you to think about a solution in several cases.

Part 1: Mixed Mode Operations

While mixed mode operations are not allowed, we still have to be careful because explicit conversions may still not produce the result we intuitively expect.

1. What would you expect the result of the following code to be just by eyeballing it?

```
a , b := int8(1), uint8(128)
fmt.Println(a  + int8(b))
```

3. Run the code
4. If you can't explain the result then try running this code:

```
a , b := int16(1), uint16(128)
fmt.Println(a  + int16(b))
```

5. Now the answer is what you would expect. Notice the difference in the code between the two example and consider how those numbers are represented in binary form. Look at what is happening in the binary representations of the numbers in each of the two snippets.
6. Hint: Try printing out the binary representations of the numbers using the "%b" directive in the fmt.Printf() command.

Discussion Question 3-1

(a) Given the different outputs in the two above code snippets, what is actually happening when Go does the conversions? (Hint: remember basic computing – signed integers are stored in twos complement form.)

(b) Doing it this way is a design decision. What other alternatives would you consider if you one of the Go designers?

7. Eyeball an estimate for what the following code would output:

```
a, b := uint8(1), uint16(512)
fmt.Println(a + uint8(b))
```

8. Now run the code and compare the actual result with what you expected.
9. Look at the underlying hexadecimal representation of the two numbers and figure out what Go is doing in the conversion. Or the binary representation.

Discussion Question 3-2

(a) Is this behaviour consistent with what you have just observed in the previous part of this lab?

(b) What does this suggest that you need to double check when you are programming in Go?

Part 2: The if Statement Sytnax

The if statement is pretty standard with a couple of variations from the usual C-style form shared by other other languages. In this exercise, you will just experiment with the examples from the lectures.

1. Working with the code from example 03-03 “The Basic if statement,” try the following.
2. Put parentheses around the test case x == 0 and see what happens when you compile the resulting code. If you are using an IDE then try it in both the IDE and editing the file in a text editor.

Discussion Question 3-3

Given the behaviour of the if statement, what can you say about the use of parentheses,

1. Undo what you did in step 2
2. Modify the code to look like this:

```
if x == 0
{
fmt.Printf("%d is zero\n", x)
...
```

3. Run the code and notice what happens.

6. Now modify the code to look like this

```
if x == 0 {
      fmt.Printf("%d is zero\n", x)
} else if x%2 == 0 {
     fmt.Printf("%d is even\n", x)
}
else {
      fmt.Printf("%d is odd\n", x)
}
```

7. Run the code and see what happens.

Discussion Question 3-4

What is the advantage to being able to declare local variables for the whole if statement?

Part 3: The if Statement Local Variables

1. Run the following code

```
func main() {
    var x int
    if x, y := 0, 0; y == 0 {
         x = x + 100
    }
    fmt.Println("Value of x is ", x)
}
```

Discussion Question 3-5

Is the output what you expected? Explain the value of x that was displayed.

Part 4: Programming Exercise 1

In this problem, we introduce one of the Go standard libraries, the os package which is defined as "a platform-independent interface to operating system functionality."

In this problem you are going to use the os.Args array to count the number of words and characters from the command line.

If your code is in the file *lab03-01.go*, when you execute the code at the command line as show below, you should see the output as depicted below.

In order to do this exercise, the following information is useful

1. You have to import the "os" package. You should take a quick look through the package to see what sort of functionality it covers.
2. The variable os.Args is a array of the tokens (words) on the command line following the name of your program file.
3. os.Args[i] is the i^{th} word – array access works just like in other C-style languages Remember that the name of the function is os.Args[0]
4. len(os.Args) returns the length of the array os.Args
5. The solution provided for the exercise uses a for loop.
6. The length of the i^{th} word can be found using len(os.Args[i])

```
[lab00]$ go run lab03-01.go this is a test
Words=4 Characters=11
```

Part 4: Programming Exercise 2

This problem is a modification of the previous one. Instead of just counting the letters, we want to return a count of the number of vowels and the number of digits.

The output should look like what is depicted below.

Some hints.

1. Each os.Args[i] can be assigned to a string variable, call it word for example.
2. You can use a for loop to iterate over the word to get each letter using the syntax word[j]
3. To test for a vowel or digit, use a switch statement rather than if statements. For example, the vowel case should look something like this

```
case 'a', 'e', 'i', 'o', 'u':
     vowels++
```

4. Notice the use of the single quotes to identify a character literal.

```
[lab00]$ go run lab03-03.go this is 327 test23
Vowels= 3 Digits= 5
```

Lab 4: Functions

The purpose of this lab is to provide you with a chance to experiment with functions and the related concepts dealing with functions in Go that we looked at in module four.

Part One: Basic Function Definitions

This first part is some experimentation with the basic function syntax. The actual functions themselves are trivial, the important point is for you to be able to see how the function call syntax works. We will start with a basic function then modify it in a number of ways to implement some of the concepts that have been presented.

1. Version One: Write a basic Go function a that takes two integers as parameters and returns their product. Wirte this version without using a named return value. This should look like the usual type of C-style function. A solution is file lab04-01.go
2. Version Two: Modify the function to use a named return value. A solution is in file lab04-02.go Experiment with a naked return versus explictly returning the named return variable.

Discussion Question 4-1

What would be the advantage of being able to use a naked return?

3. Version Three: Modify your function to now return both the product and sum of the two parameters. Do not use named return values. A solution is in the file lab04-03.go
4. Version four: Modify version three to use named return values. A solution is in the file lab04-04.

Part Two: Using Defer

In this section, we will be exploring the defer operator. Consider the following code:

```
func first() {
    defer second()
    fmt.Println("first")
}
func second() {
    fmt.Println("second")
 }


func main() {
    defer first()
    defer second()
 }
```

What would be the result of running this code? The code is the file lab04-05.go. Run it and see if your solution is correct.

Now change the main function to be the following:

```
func main() {
    first()
    defer second()
}
```

Again, what do you think the output will be? Make the code change and run it to see if you are correct.

Part Three: Varadic Functions

1. Write a function that finds and returns the minimum and maximum values out of a list of ints. Use a Varadic parameter. If list of its is empty, return a boolean test value to indicate an empty list
2. You should be able to execute the function like this

   ```
   min, max, empty := limits(1,2,3,4,5,6)
   ```

   where min and max are the corresponding values and empty is the bool test value
3. Test your function to see that it is working correctly on a variety of values
4. Hint: Use the range function to iterate over the list. Also remember that if you use named return values, they are initialized to their default zero values. Also you might try using a switch statement instead of a series of if statements. A solution is in the file lab04-06.go

Part Four. Recursive Functions

We saw recursion being used to sum up a series of numbers. Now using that as a guide, write a function that computes the nth Fibonacci number using recursion. A solution is in lab04-07.go

Part Five: Responding to an error

To do this problem, we will use the "'http" package to access a web page and print out the status code of the http response if the page was accessed.

1. To do this, you need to import the "net/http" package.
2. You can perform a get operation a webpage with the command

   ```
   resp, err := http.Get(URL)
   ```
3. The URL is a string like "http://www.Capitalone.com" – the protocol portion of the URL must be included like in example here
4. The first parameter returned is the http response object. The second is an error object that is non-nill if the fetching of the URL failed.
5. If the Get command was successful, then there should be a string with the returned status in the variable resp.Status (assuming that resp is the variable you assigned the response value to).
6. Write some code using these commands to get a common website (like google) and print out the status code.
7. If the fails, then print out what was returned as the error. Remember if there was no error then the error value returned is nil
8. A solution is available in the file lab04-08.go

Lab 5: Arrays and Slices

The focus of this lab will be on understanding the behavior of arrays but with an emphasis on slices since they are used more often in Go.

Part One: Passing Arrays and Slices to Functions

Version 1: Arrays as parameter

1. Write a function named swap that looks like this.

   ```
   swap(a [2]int)
   ```

   The function takes an integer array of two ints as parameter and swaps the position of the two elements. Remember that you can use parallel assignment.
2. Demonstrate that the array in the calling function is unaffected by the action of swap(). Remember the array is passed by value so swap() is working with a copy of the array from the calling function. A good way to demonstrate that you are working with a copy is to print out the address of the first element of the array both before and after the swap function is called and of the array used in the swap function. (&a[0]). You should see that the address of the array in the calling function and the array in the swap() function are different.
3. A solution for this is in file lab05-01.go
4. Can you pass any other type of array to swap? Like a [3]int array? Why or why not?

Version 2: Pointer to array as parameters

1. Change the swap function to take a pointer to an array of type [2]int and try running it again. Make sure to pass the address of the array to the function, not the array itself. Examine the results.
2. A solution to this is in the file lab05-02.go
3. Will this version work with the same array if the function parameter is a pointer to a [3]int? Why or why not?

Version 3: Slice as parameter

1. Change the array you are using to a slice.

   `ie.  []int{1,2} instead of [2]int{1,2}`
2. Now change swap() so that it takes a slice of ints as a parameter instead of a pointer to an array and rerun the function. Remember that you don't need to use pointer notation for slices. A solution is in file lab05-03.go
3. Notice that it works the same as the array pointer example.
4. Change the slice so it has a length of 3, then see if the function still works. Why or why not?

Part Two: Reversing a Slice

1. Write a function that takes a slice of ints and reverses the order of the elements in place (ie. changes the original slice in stead of creating a reversed copy.)

   `s := []int{1,2,3,4,5}` becomes `[5,4,3,2,1]` after call to reverse()
2. Hint. Parallel assignment makes this a lot easier. Use a "for" loop, set one index to the start of the slice, and another to the end of the slice. Then swap the elements at those positions. Increment the start position by 1 and decrement the end position by 1. Continue as long as the end position is greater than the start position.
3. A solution is in file lab05-04.go

Part Three: Reversing an Array

1. Using the reverse() function you wrote in the previous part, use it to reverse an arbitrary array of ints. No changes to the function are necessary. The array should be reversed in place.
2. Hint, do not work directly with the array but use a slice.
3. A solution is in file lab05-05.go

Part Four: Equivalence of Slices

1. Consider this code that is in file Lab05-06.go

```
a1 := [2]int{1, 2}
a2 := [2]int{1, 2}
fmt.Println("a1 == a2?", a1 == a2)
original := []int{1, 2, 3, 4, 5, 6, 7}
s1 := []int{1, 2, 3, 4, 5, 6, 7}
s2 := original
fmt.Println("original and s1 ", &original[0] == &s1[0])
fmt.Println("original and s2 ", &original[0] == &s2[0])
```

2. Run the code and explain the results. The following discussion may help.
3. Remember that == is defined on arrays if they are the same type and their corresponding elements are equivalent. However slices are pointers so Go does not allow us to do the same sort of comparison on pointers as we do with arrays. Change a1 and a2 to slices and try to run the same code and you will get a compiler error

   One problem is trying to sort out what happens when we have two slices and assign one to another. As we can see if we compare the addresses of the first elements of the slices, we just get another pointer to the same underlying array. So then what would it mean for us to say that for two slices s1 and s2 the expressions s1==s2 should mean? This is the old shallow versus deep comparison problem.
4. Instead, write a comparison function called equiv(s1,s2) that takes two slices s1 and s2 of the same type (like []int) and declares then to be content equivalent using the same logic as for arrays.
5. Modify the function equiv(s1,s2) so that the requirement that the two slics have to be the same length is dropped. Now only compare only up to the length of the smaller of the slices.
6. A solutions are in the files lab05-07.go and lab05-07-2.go

Part Five: Copying a slice

1. Go has a copy function for slices that allows you to copy the contents of one slice into another with the syntax copy(destination, source) as long as the two slices are the same type. The copy will terminate when the end of one of the slices is reached. An example of the copy utility is illustrated in file lab05-08.go
2. Write a function called mkcopy(s) where s is a slice and the function returns a copy of the slice s. For the purposes of this exercise the function only has to work with slices that are of type []int. Hint: the function should use the make() command and return a new slice.
3. A solution is in the file lab05-09.go

Part Six: Multi-dimensional Arrays

1. For this we are going to have a simple structured representation of a text. A text will consist of one of more paragraphs, and a paragraph will consist of one or more sentences and each sentence will be a slice of strings which are the words in the sentence.
2. Use the simple text in the file lab05.txt file as your test data or any other text you want.
3. Use a multidimensional array literal to initialize the text object
4. Write a function to count the number of words in the overall text.
5. A solution is in the file lab05-10.go

Lab 6: Maps (Word Count Project)

This lab is going to be a change from the other labs and more of a project. We are going to create an application that takes a text file (in this case one of Winston Churchill's famous speechs) and produce a word count frequency. As output we will get a list that looks like this

```
words              count
  10                 3
   8           10
   3           21
```

This means that 10 words in the text occurred 3 times each, 8 words in the text occurred 10 times each and 3 words occurred in the text 21 times each.

In order to solve this problem, we are going to break it down into steps and solve each step in turn. In order to produce solutions at each step, we will use functionality from several different Go packages.

The steps are:

1. Read the text from a file
2. Convert the contents of the file to a string
3. Remove punctuation (“there.” and “there” should not count as two different words)
4. Remove capitalization (“Hi” and “hi” should not count as two different words.)
5. Split the long string of characters into individual words.
6. Count the number of times each word occurs.
7. Sum up the frequencies and output the results.

Part 1: Reading from a file.

Go has a number of file utilities that we can use for reading and writing to files. The one we are going to use in the package “ioutils” and is called Read(). At this point you should go and use the online documentation to understand what this function does.

Our first function will be called myread(filename string) and given the name of a file, the function copies the contents of the whole file into a variable that I will call rawtext.

1. Because any file read may not work for a number of reasons, an error is also returned. If this error value is non-nil, we want to panic and end the program.
2. The rawtext variable is of the type []byte which is called a bytes slice. All this does is store the raw contents of the file as an array of bytes in memory once they have been read in.

3. The first task is to write this function called myread() which opens the file when passed the name of a file. The file provided is called “churchill.txt” and is in Mod06 lab directory. The function myread() will read the file content into the rawtext variable If there is an error, the function should print out the error and then cause a panic to shut down the application.
4. The function should return a string, use a named return variable so it compiles until we add the byte to sting conversion code. We wont actually return anything other than an empty string in this first iteration of the code.
5. You have to import “io/ioutils” to be able to use the Read function
6. Add a temporary fmt.Println(rawtext) so you can see the file contents in raw bytes.
7. A solution is in file lab06-01.go

Part 2: Converting to a String

1. A string is just a way to interpret bytes. There are a number of rather sophisticated ways of working with UTF-8 and runes, but for now we will keep it simple. We want to let Go handle the conversion for us so we will just ask Go to interpret the rawtext variable as string. To do this we just need to take a slice of rawtext and convert it into a string. Remember how to convert data types from module two. Make sure that myread() now returns the contents of the file as a string.
2. The main function should call myread('churchill.txt) and get back a string containing the speech that was in the file. Print out the results of the function call to make sure that this is working properly
3. A solution is in file lab06-02.go

Part 3: Removing Punctuation

1. In order to do this we will need to use some utilities that work on strings form the “strings” package. The one we will uset is the strings.Replace(str, s1,s2, count) which replaces occurrences of sub-string s1 in string s with s2 for as many times as count specifies. If count < 0 then it replaces all of them.

2. There are three punctuation marks “.,;” in the text so we can loop through that list of punctuation marks and apply the strings.Replace() replacing each punctuation mark in turn with the empty string. So this would require a for loop and the range function and a string of punctuation marks to iterate over.

3. To do this, we need to create a function called strproc that looks like

```
strproc(s string, punc string) string {}
```

where s is the input, punc is the string of punctution marks, and the return value is the string with all of the punctuation marks stripped out.

4. One of the problems is that as we iterate over the punctuation marks is that the individual values plucked off the string are not strings of length 1, they are characters or runes. In order to get the Replace() function to work, those punctuation marks have to be converted from runes to strings. Again, remember module two about how to convert data types.
5. We will Print out your result to make sure your function is working correctly.
6. A solution is in file lab06-03.go

Part 4: Breaking the string into words

1. Now we need to lowercase all of the letters. We can use strings.ToLower() for that. Look up how to use it in the Go language documentation.
2. We also need to split the big long string into a series of words There is is another function in strings called strings.Split() that will do that for us. For these two, you should be able to look up easily how to use them.
3. Add this functionality into the strproc() function. But now we have to change what the function returns since it returns an array of strings instead of just a string. Since we do not now how long our list of words will be, we should return a slice of strings ([]string)
4. Make sure you print out what the function returns to make sure it is working correctly. There is a solution in file lab06-04.go

Part 5: Making the map

1. Now we need a function to load the list of words into a map.
2. Create a function called makemap(words []string) that takes a slice of strings as a parameter and returns a map of type map[string]int.
3. The key for each element will be the word in the text and the value will be the number of times that the word occurs in the text.
4. You can use the fact that if the word is not in the map yet, when you try to access it you will get a value of zero for it's occurrences
5. Use the make() function to create the map.
6. Once the map has been returned to the main function, print it out to make sure everything is working.
7. A solution is in file lab06-05.go

Part 6: Remapping

1. We now have to create a list of words for each frequency so that we can compute some statistics. We could do this in one step, but this is a bit more fun.
2. What we want to do is to combine elements in our map so that

   { struggle :1} and {given : 1} are now represented by { 1 :[struggle, given]}
3. We want to create a new map of the form count[int][]string, where for each word frequency, we now have a slice of words that have that frequency.
4. Create a new map like this one using make

   ```
   count[int][]string
   ```
5. Then we iterate over our existing word map.
6. When we get a value {word : n} then we update the entry in count like this

   ```
   count[n] = append(count[n], word)
   ```
7. Notice that the all of the words in each slice are unique.
8. Implement a function that does this and that looks like the following

   ```
   func transmap(words map[string]int)
                  (count map[int][]string) {}
   ```
9. Test your function to see that it works. A solution is in the file lab06-06.go

Part 7: Sorting and printing the results

1. First we are going to sort the results. We are going to do this by extracting the keys from our count map we just made and put them into an int slice, sort the slice, then iterate over the count map in the order of the sorted keys, add up the number of words in each slice for that element and print the results.
2. Create a function called results() which take our count map as a parameter.
3. To sort your slice of keys, import the “sort” package and use this line.

   ```
   sort.Sort(sort.Reverse(sort.IntSlice(keys)))
   ```
4. After this executes, the keys slice will be sorted the way we want it. We will revisit this line after we look at interfaces to see what happened.
5. Now we can iterate through the map on the this keys slice and just take the length of the value of each element and print out the result.
6. There is a solution in file lab06-07.go

Lab 7: Advanced Functions

The problems in this lab follow fairly closely the ideas presented in the student manual. The important points of the lab are that you understand what we are doing with the functions when we treat them as first class objects.

Part 1: Function map

For explanatory purposes, suppose we have a function of type 'func(int) int', lets use square(int) that returns the square of its argument as an example of a function of this type.

A function map would apply the function square(int) to a slice of ints say [1,2,3] and return a slice of results [square(1), square(2), square(3)] or [1, 4, 9]. This map operation on functions is a common functional programming technique.

In this part of the lab, we will emulate a function map operator.

1.1 Emulating the map operator.

1. For this exercise, we will write a function mapf with the signature

   ```
   mapf( f func[int]int, arg []int) []int {...}
   ```

2. The function mapf will emulate the behavior of a function map.
3. Hint: You want to iterate over the slice args of ints and build up a new slice of results by inserting the result of f(arg[i]) into the i^{th} position of the results slice.
4. Use two functions to test it. Suggestions are a function square(iint) that returns the square of an int and a function neg(iint) that returns the negative of an int. These are simplistic but we just want to be able to work with several different functions of the same type.
5. A solution is available in file lab07-01.go

1.2 Emulating an in place map operator

1. In the previous problem, your mapf function returned a slice of results from applying the function. Now change that function to a new form of map, call it mapf_r, that does what we call "in place" operation. Instead of returning a new slice, it replaces the values in the input slice with the results of applying the function.
2. There are only a couple of places where changes have to be made.
3. The new map function should have the signature

   ```
   func mapf_r(f func[int]int, arg []int) {}
   ```

4. There should be nothing returned, the function alters the input parameter slice of ints directly.
5. A solution is provided in the file lab07-02.go

1.3 Reversing the problem

1. Now we will do the opposite. Given a slice of functions, apply them all to an argument and return a slice of results. For example, suppose we have the square(int) and neg(int) functions as described earlier.
2. We defined a function with this signature

   ```
   func fmap(functions []func(int)int, arg int) []int { }
   ```

3. Then given a slice of functions like

   ```
   funcs := []func(int)int{square, even}
   ```

4. When we call

   ```
   fmap(funcs, 2)
   ```

   then we get back the slice [4, -2]
5. A solution is provided in file lab07-03.go

1.4 Function literals

1. Modify the solution to the preceding part so that we do not need to have actual functions named square() and neg(), we can just use function literals instead.
2. Hint: this may seem challenging but all you need to do is replace the use of the function names “square” and “neg” with the appropriate function literal.
3. A solution is provided in file lab07-04.go

Part 2: Dispatch Function

A dispatch function is a function that executes a different version of itself whenever it is called depending on the status of some condition at the time it is called. The term dispatch is used to describe how the function being called “dispatches” the function call to one of many possible function implementation, just like a taxi dispatcher sends a request to any one of a number of cabs.

Dispatch functions can be implemented in many ways, for example, C uses function pointers and some OO languages use some form of polymorphism. At its core, a dispatch function is a big switch statement that manages the dispatch logic as to what gets called. Our form will be a bit clunky but there is a way to clean it up nicely which we will look at later on.

2.1 Basic Dispatch Function

1. Our dispatch condition will be a package level variable called status which is just an integer. This will represent a sort of count of things that have gone wrong. The status variable is initialized to 0.
2. Our dispatch function will be called oops() that will be called by us to emulate something going wrong.
3. Depending on the value of status, oops() will return a different function to be executed. Each of the functions returned will increment the value of status and print out a message.

   status 0: prints out “Everything is cool”

   status 1 or 2: prints out “Things are heating up”

   status 3: prints out “Too hot for me” and then calls a panic() to end the application.
4. The oops() function will essentially be a switch statement that will return a function literal for each of the case. The function literal implements the functionality for that case.
5. Because oops() returns a function, we still have to execute the function that is returned. This means that your call to oops will look like this “oops()()” which means “call oops, then call the function returned by oops”
6. The function oops() will look like

   ```
   func oops() func() {..}
   ```
7. Test your oops() function in a loop like this

   ```
   for i := 0; i < 5 ; i++ { oops()() }
   ```

   just in case your panic doesn't work.
8. A solution is in the file lab07-05.go A cleaner version is in lab07-05-1.go

2.2: Modifying the Dispatch Function

1. Modify the dispatch function oops() you wrote in the previous problem so that instead of returning a function, the function literals are executed right in the switch statement instead.
2. This means that the signature of oops is now

```
func oops() {..}
```

3. And we can change oops()() to oops()
4. Hint: Remember that func() {…} is an anonymous function and func() {..}() executes it.
5. A solution is in file lab-07-06.go

2.3 Creating a Closure

1. While we have been working with anonymous functions, we have not created any closures even though each of the anonymousr functions references the free variable 'status'. For this problem, go back to the version of oops() that returns a function (lab07-05.go)
2. How would we create a closure f so that every time we called f() we would call oops but with a local closure copy of the free variable status? We would have to create an instance of that function that is returned by oops() and then call that instance. Look at the example in the student manual fo a closure to see how it works and create a closure of the function returned by oops().
3. A solution is in file lab07-07.go

Lab 8: Structures

This lab will have two programming problems that not only use structs but require you to use some of the knowledge from the previous modules.

Part 1: Building a Data Frame

Data frames are common objects in a variety of statistical languages and packages. Basically a data frame is a square array of data with column headers and often a collection of statistics that are computed for for the embedded data table. For our lab we will make a couple of assumptions, for example, we will assume the data table is always square and that all columns must have headings.

For our data frame, we are going to keep four statistics for each row of the table

1. The number of entries
2. The mean of the value in the row
3. The maximum value in the row
4. The minimum value in the row.

The data frame itself will also keep these statistics but for the whole table (ie. The mean for the frame is the mean of all the data entries in the data table). The data frame will also remember number of rows in the data table.

We will build up this problem in chunks like we did in module 6 with maps.

1.1 Defining a row struct.

Since a row is data structure itself, it needs to be defined as a struct. The one used in the solution is

```
type row struct {
    data     []int
    n        int
    mean     float32
    max, min int
}
```

1.2 Defining a Construtor

Because there is more to creating a row than just copying values, we have to do some computation, we need a constructor. Write a function that given a slice of ints, returns a pointer to a row struct, with all of the fields properly initialized.

The signature of the constructor used in the solution is

```
func makeRow(d []int) (s *row) {}
```

To keep things modular, you might want to start with a helper function that does does the computation for a slice of ints. Something like this

```
func rowProc(d []int) (max, min, n int, mean float32) {}
```

To keep the problem simple, if the constructor is passed an empty slice, then it should just return a nil pointer.

1.4. Write a print function

To see how your constructor works, write a printRow(r *row) function that pretty prints your row so you can see that your constuctor is working. There is one in the file row.go

1.5 Defining a Frame

The data frame itself will be a struct of rows and other statistical information.

The definition used in the solution is found in frame.go in the directory lab08-02 and looks like this:

```
type frame struct {
    headings   []string
    rows, cols int
    data       []row
    max, min   int
    mean       float32
  }
```

1.6 Building the Frame

To create a frame, we will pass it a slice of rows and a slice of strings to act as column headings. It is possible to also write a constructor that creates a frame directly from a a square array of ints, but for the purposes of our lab, this is probably an easier approach to the problem. The constructor for the frame should look like this:}

```
func makeFrame(h []string, d []row) (f *frame) {}
```

Some points to watch

1. It is probably easier to compute the frame stats right in the constructor rather than in a helper function.
2. The rows that we are passing are independent objects. We do not want changes to those rows in other places to affect the data in our data frame so we need make sure we copy the rows into the frame. Remember that we are passing a slice of rows, which are just like a pointer.
3. The constructor for the row returns a pointer to a row. The constructor for the frame takes a slice of rows, not a slice of pointers to rows. This means that if you defined two rows like this:

```
r1 := makeRow([]int{1, 1, 3})
r2 := makeRow([]int{9, 102, 5})
```

Then r1 and r2 are pointers to rows, not rows. That means when you pass them to the frame constructor, you need to dereference them like this:

```
d := []row{*r1, *r2, *r3} // a slice of rows
h := []string{"One", "Two", "Three"}
f := makeFrame(h, d)
```

1.7 Write a print function

Just as we did with the the row struct, write a function that pretty prints a frame to make sure that it is working.

Part Two: Linked List of Integers

This problem shows how we can use pointers and structs to create a very easy implementation of a linked list. One of the goals of this exercise is to give you some familiarity with using pointers in Go. The content of the list is not important so we will just be using a linked list of integers.

2.1 The Node and List

Our node are going going to be simple. The basic definitions of the node and list structures appear in node.go in directory lab08-03

```
type node struct {
    prev     *node
    contents int
    next     *node
  }
```

The ints are the payload which is not important in this problem, but the forward and backward links are pointers to nodes. The list is even simpler, it is just a struct with a reference to the start node and to the end node

```
type list struct {
    start *node
    end   *node
  }
```

2.2 Implementing the functionality

We are only going to partly implement the linked list in this lab. The actual code will be left here as a challenge but the complete solution is available in directory lab08-03. For our partial implementation, write the following functions:

1. Write a function that creates an empty list.

   ```
   func makeEmptyList() *list
   ```

2. Write a function that given an integer and a list, adds the integer to the list as a node.

   ```
   func addNode(f int, l *list)
   ```

3. Write a function that prints out the list from the start to the end.

   ```
   func printList(l *list)
   ```

Lab 9: Methods

This lab will build on the previous lab and move towards a more object oriented type of implementation of the previous problem.

Part 1: Creating Row Methods

In the previous lab, we created a data frame structure along with a set of functions that operated on the data frame and the individual rows. It is a lot more natural to think fo these functions as something we apply to the data frame.

For the first part of the lab, we will rewrite some of the functions as a method set for a row object. We will leave the makeRow() function signature as is since it doesn't actually operate on a row object

1. Rewrite the procRow() function and a method on a row called computeSt ats() with the signature

   ```
   func (r *row) computeStats()
   ```

   Since we are using a row object as a receiver, we can access all of the internal structure of the row without having to pass data as parameters or worry about return values. Making this into a method now means that we can actually recompute the row statistics anytime any row data values change.
2. Rewrite part of the makeRow() function body so that it now uses the new method computeStats() instead of the procRow() function.
3. Rewrite the printRow() function as a method with the following signature

   ```
   func (r *row)printRow()
   ```
4. Test your code to make sure it runs.

Experimenting with Code

1. Change the printRow() method to have the signature

   ```
   func (r row)printRow()
   ```

   Try to compile and run the code and explain the result.

2 Try compiling both printRow() methods, one with a pointer receiver and one with a struct receiver and explain the results.

A full solutions is in the directory lab09-01

Part 2: Creating Frame Methods

This lab will do the same for frames that we just did for rows, converting the functions into frame methods.

1. First create a method that adds a vector of strings as headings with the signature

   ```
   func (f *frame) addHeadings(h []string)
   ```

2. Create a method that adds a row to the frame with the signature

   ```
   func (f *frame) addRow(r *row)
   ```

   Remember to add the row only if it has the same number of columns as the rest of the frame.

3. Add a method called computeStats() that computes the overall satistics for the frame with the signature (refactor the existing function)

   ```
   func(f *frame)computestats()
   ```

4. Add a method called printFrame (refactor the existing function)

   ```
   func (f *frame)printFrame()
   ```

5. By adding a method in the above points, what is meant is that the functions that we wrote in the last lab should be converted to methods.

6. Notice that we have duplication of method names for rows and frames, but this not a problem because they duplicate names are in different method sets.

Something that you should have noticed as you refactored the functions into methods is that the code became cleaner. The reason for this is that a lot of the parameters needed in the body fo the method are now accessed through the receiver instead of being passed in parameter lists and return value lists.

A full solution is available if the directory lab09-02.

Lab 10: Interfaces

This lab is based on a classic Go example that a number of authors have used to illustrate how interfaces in Go are used for abstraction. We are going to start with a fairly common process in OO which is the abstraction mechanism of generalization – where an abstract type is created to represent a grouping of concrete types that have some form of similarity to each other that we want to capture.

We are going to create four types: a square, a rectangle, a triangle and a circle and then construct two abstract types; a general shape type called shaper and a quadrilateral type called quadder. These last two will be the abstract classes in our lab exercise.

Part 1: Create the types structs.

All fields should be of type float64

1.1 Define a struct for each type.

1. The square should have a field to represent the length of a side
2. The triangle should have two fields representing the base and height
3. The rectangle should have two fields for width and height
4. The circle should have a radius field

1.2 Add some methods

1. Each one of the types should have an area method that computes the area of the object as a float32. .
2. For the circle, add a method that computes the circumference of the circle
3. For the rectangle and square also include a method called diagonal() which returns the length of a diagonal for the shape.
4. Notice that for the circle, you will have to import the value math.Pi
5. For each of the objects, create a factory that creates objects of that type.

A complete solution for this part is in the folder lab10-01.

1.3 Test your objects

1. Test out your code to make sure that you can create objects of the given types and that the methods all work correctly.
2. A full solution is in the folder lab10-01

Part 2: Create an abstract type and variable

1. In order to create an abstract type, define an interface like this

```
type shaper interface {
     area() float64
}
```

   All of the objects defined so far are now all of type shaper because they implement the shaper interface, and they implement this interface by virtue of having a method that has the signature

```
     area() float64
```

2. Also define an interface like this

```
type quadder interface {
     diagonal() float64
}
```

   And now squares and rectangles are quadders because they implement the method specified in the interface. However squares and rectangles are both shapes as well. Circle has a method that is not in the shaper interface, but this in no way inhibits it from being of shaper type.

3. Create a variable of type shaper. Notice that you can assign an object of any of the types we have defined to that variable. And we can call the area() method on whatever we have defined.
4. Create a variable of type quadder and notice that you can assign a square or rectangle object to it, but not a circle or triangle.
5. However notice that you cannot assign a circle to a varaible of type quadder.
6. A full solution is in the folder lab10-02

Part 4: Collections of type shaper

The usefulness of the abstract types starts to be apparent when we can treat circles, squares and the rest all being of the same abstract types shape.

1. Create a slice of shapers. Add one of each shape to the slice.

```
     x := []shaper{makeCircle(1.0)…..}
```

2. Loop through the slice and call the area() method on each shaper in turn. Notice that the right method is called. Shaper is polymorphic, it calls the right method depending on the concrete type (circle, square, etc) that is actually referenced by the variable when the method is sent.

3. A full solution is in the folder lab10-03

Part 5: Access the concrete types.

In this section we access the concrete type methods for the objects in the collection in the previous section. While looping through the collection of shapes, use a switch statement to test the types of the object. There should be three cases to the switch.

1. If the object is a circle, then the circumference should be printed out.
2. If the object is a quadder, then the diagonal should be printed out.
3. If the object is anything else, then name of the type should be printed out.

The full solution is in file lab10-04.go

Part 6: The empty interface.

1. For the last section, define an interface type of “all” which is the empty interface.
2. Create the collection as in the previous section but in addition to shapes now add a few elements of other types such as an int or a float32 for example.
3. Ensure that the switch still works as it should.

Lab 11: Concurrency

In this lab we are going to work with two problems that demonstrate some basic concurrent thinking.

Part One: The Countdown

This particular exercise is a modification of Brain Kerningan's example. We want to write a program that simulates a countdown with the ability to abort the countdown

Step 1: The Basic Countdown

1. Write a first cut at the countdown loop.

```
func launch() {
     fmt.Println(“**BOOM**”)
}
func main () {
     fmt.Println("Commencing Countdown.")
     for countdown := 10; countdown > 0; countdown-- {
          fmt.Println(countdown)
     }
     launch()
}
```

2. Create a Go file, enter this code and run it and see what happens. This code is in lab11-01.go
3. The problem is that if we just use a loop to countdown, then we will finish the countdown in under a millisecond. We need to be able to force the countdown loop to do one iteration per second.
4. To do that, we will feed it “ticks” at the rate of one a second. To do this, we will use the time.Tick() function which returns a channel that sends timestamps at a regular interval. We will create the “ticker” by adding the line below before the for loop

```
tick := time.Tick(1 * time.Second)
```

5. Tick is a channel that is feeding data to us at one second intervals. We are not intersted in the data so we read from the channel in each iteration of the loop. Of course we will block until the next tick is available so the loop will proceed at a pace of one iteration per second. We want to place the line below in the loop.

```
<- tick
```

6. Notice that we are not saving the value, we are just reading off the channel. Since there is one input per second, this read blocks until the tick is there.
7. Make the changes and run the file, you should see a regular looking countdown. A full solution is in file lab11-02.go

Step 2: Adding an abort

1. Countdowns can go wrong so we need an abour facility which will be done be pressing the enter key on the keyboard.
2. This is clearly a concurrency issue. The keybord has to be monitored as an independent task from the countdown.
3. We can add an abort go routine that looks like this:

```
abort := make(chan int)
go func() {
     os.Stdin.Read(make([]byte, 1))
     abort <- 0
}()
```

4, The Stdin.Read() method blocks waiting for a return to take place at the keyboard. One a byte has been read after return is pressed, it sends a message on an abort channel which we will add next.

5. Add this code above just after the start of the main() function so that the first thing done before the countdown starts is to set up the abort channel. A full solution is in lab11-03.go

Step 3: Multiplexing

1. The problem is that no one is listening to the abort channel. What we want to do is multiplex the abort channel with the tick channel. If we get an abort between ticks, then we want to exit the program.
2. The multiplexer we need is the select statement. The abort channel will be one case and the tick channel the other. Each time through if there is no tick yet, the abourt channel has a chance to be read.
3. In the for loop we need this select statement:

```
for countdown := 10; countdown > 0; countdown-- {
     select {
          case <-tick:
               fmt.Println(countdown)
          case <-abort:
               fmt.Println("Aborted")
               return
          }

     }
```

4. Add this code and test your countdown. The full solution is in lab11-04.go

Part Two: The Sieve of Eratosthenes

This is a very famous algorithm for finding primes. This is a very famous implementation of the algorithm in go using a concurrency patterns called daisy chaining. Basically we will create a goroutine called a generator that will send integers over a channel, A goroutine will filter out all the numbers divisible by two then send the results to a next goroutine which will filter out of those all that are divisible by three and then pass it on to another goroutine.. and so on.

Step 1: Create a generator

The first thing we want to do is have a sequence of integers to work with. To do this, we create a generator function that just puts integers one after another onto a channel. The code for this looks like this:

```
func Generate(ch chan<- int) {
    for i := 2; ; i++ {
         ch <- i // Send 'i' to channel 'ch'.
    }
  }
```

Write and test this goroutine to make sure it works and that you understand its output. A solution can be found in lab11-05.go.

Step two: Create a filter.

Now we add the filter gogoutine which looks like this:

```
func Filter(in <-chan int, out chan<- int, prime int) {
    for {
         i := <-in // Receive value from 'in'.
         if i%prime != 0 {
              out <- i // Send 'i' to 'out'.
         }
    }
  }
```

The actual internal logic of the filter is quite simple.

To test this goroutine, create a simple filter that takes 2 as the prime and produces a stream of odd numbers. For testing purposes, put this in a loop that terminates after a particular value has been reached in the filter output. If you see some of the numbers missing from the output stream, it is because the output statement is working a bit too slowly. You can fix this by putting a brief sleep statement just after the output statement to slow the loop down.

A solution is availabe in the file lab11-06.go

Step Three: Daisy chaining

In this step, since our generator starts sending numbers at 2, the first number on the channel will be 2 or the first prime.

```
go Generate(source)
prime := <-source
```

Now we can make a filter to filter all the divisible by two numbers out.

```
filtered := make(chan int)
go Filter(source, filtered, prime)
```

In order to create the sieve, we want to take the output of this filter and filter it to throw out all of the numbers divisible by 3. To do this we repeat what we just did but now the source comes from the filter we just created instead of the generator.

```
source = filtered
```

All of this now can be put in a loop that repeats this process until we decide we have enough primes, in this case 10.

```
func main() {
     source := make(chan int)
     go Generate(source)
     for i := 0; i < 10; i++ {
          prime := <-source
          fmt.Println(prime)
          filtered := make(chan int)
          go Filter(source, filtered, prime)
          source = filtered
     }
}
```

A solution is available in lab11-07.go

Lab 12: Testing

Part One: Unit Testing

For this section we are going to test a function that checks to see if a string is a palindrome or not. This exercise has been presented in various ways by a number of authors.

Step 1: Write a palindrome function.

1. To test your coding, write a function that takes a string as input and returns a true if it is a palindrome and false if it is not. If you are having problems, there is one already provided for you in lab12-01.go
2. Create a lab12-01_test.go file and put two test functions in. These are our first two test cases. The first function should be a palindrome and the second not a palindrome. Remember that each test function is a test case. A solution is in file lab12-01_test.go
3. Use the "go test" tool to test your function.

Step 2: Add more test cases

1. Lets add two more test cases, the string “été" and the phrase “Adamada”
2. Run the tests and see if you can explain the results
3. A solution is in directory lab12-02

Step 3: Finding the bugs.

1. Do not read this if you want to debug the code yourself.
2. Hint 1: strings in Go are utf-8 with variable width characters.
3. The current code goes through the strings byte by byte when we should be going though code point to code point. So we have to convert the string to a slice of runes. The conversions you need are in the "unicode" package.
4. Hint 2: In order to be equivalent, two letters need to be the same case.
5. Try re-writing the isPal() function your self to make it pass. A solution is in the directory lab12-03

Part Two: Benchmarking

Since the types of results that experimenting with benchmarking will vary from machine to machine, for this part of the lab, try the following two experiments

1. Try running the benchmarks for the Fibonacci function but increase the size of the target number and observe the changes in benchmarking results.
2. Try point 1 again but this time with the the profile option

   go test bench=. cpuprofile=cout.log

   and examine the output with the go tool pprof

Lab 13: Packages

There are no labs for this module, aside from seeing if you can use go get to install a remote dependency on your local machine. Check to see what is available to you inside the Capital one firewall.

Lab 14

This lab is a mini-project which should incorporate most of the topics in this module as well, since this is the last lab, a few additional wrinkles that may require you to consult the golang documentation.

The exercise is to write a utility program for a company that updates their current customer list. When a customer is created, a customer file is created with the customer name, their newly created customer id and their contact phone number. The content of these files is then added to the master customer list.

The Go program you will write in this lab will do a number of things:

1. Prompts the user for member name and then opens a member data file based on that name.
2. The contents of the file are read.
3. The member id is validated with a validation algorithm.
4. The last four digits of the phone number are masked with ****.
5. The resulting data is added to a master member file.

Part 1 – Getting the member file name and open the file.

1. In the lab 14 directory are three customer file named godel.dat, .dat and bohr.dat. There is also file called members.dat that already has some data in it. You should examine the contents of the files so that you know what the data is you will be working with. The file names will always be in lower case ASCII alphabetic characters.
2. Write a program that prompts the user of a user name. Is the supplied name has a corresponding dat file (i.e. the user supplies “bohr” and the program should look for the file “bohr.dat”). Is the corresponding file does not exist, then the user should get an error message and be prompted to re-enter the name. If the user enters a single carriage return then the program terminates. The name entered by the user can be in any case and should be converted to lowercase before trying to open the file. There are no embedded spaces in the file name.
3. If the file exists, then it should be opened. If the file cannot be opened for any reason, the user should be prompted to re-enter the name.
4. Make sure that you save the name the user typed in because you will need it in one of the following sections.

A solution to the first part in is the file lab14-01.go. Your solution may be different but as long as it does

Part 2 – Validating the user name.

1. The first line of the user.dat file contains the user name. The rule is that the name in the file should match the name on the file. If the file does not match, then an error should be printed and the program exited.

2. The problem is that the name in the file may have accented characters while the filename does not. The rule is that all accents should be ignored. This means that we have to do a regular expression replacement of all accented vowels with their unaccented equivalents before we do the comparison.
3. To make it simpler and to avoid a lot of typing, we will assume the only accented characters that we will ever find are [è, é, á, å, ö, ó] which should be replaced by [e, a, o]. The way to do this is to use a regular expression to do the replacement on the string with the accented characters.
4. It is suggested that you write a function validate(string, string) that takes the two different forms of the name and returns a Boolean which is true if they match.

Part 3 – Customer ID validation and phone number masking

1. The second line is the customer ID. As a check, the value must be divisible by 23.
2. Write the code to check that this number is divisible by 23, and if it is not, print out an error message and exit.
3. The third line is a phone number, write the code to replace the last four digits of the phone number with asterisks so that the number looks like: (123) 456-****

Part 4 – Updating the customer.log file

1. There is a customer.log file in the directory lab14 resources directory with some data in it. Look at the format of the entries.
2. Write a method that takes the data that you have processed from each customer ".dat" files and use it to add a new line to the customer.log file – specifically you should add three lines.
3. This is a bit more challenging since you now have to open the existing file in append mode. You should be able to find out how to do this by consulting the golang "os" package documentation.