CAS CS 350

Lab 1

1. Packages
    a. Every Go program is made up of packages
    b. Every Go program should contain a package called main
    c. Package statement should be the first line off any go source file
    d. Programs start running in package main
    e. Entry point of a Go program should be the main function of main package

```
// name of the source code file is main.go (it could be whatever you like)
// this package is called
main package main
// Entry point of a Go program is main.main i.e. main function of main
package func main (){
    // println in built function is called
    println("Hello from Go")
    // prints Hello from Go in console
}
```

    f. To compile and execute the go program use go run command
    g. Packages could be imported via import statement

```
// this package is called main
package main
// fmt package contains methods to interact with console like Print and Scan
import "fmt"
// Entry point of a Go program is main.main i.e. main function of main package
func main (){
    // Println method of package fmt is called
    fmt.Println("Hello from Go")
    // prints Hello from Go in console
}
```

2. Import
    a. Fmt package – interacting with console (fmt package includes functions related to formatting and output to the scrren)

        Import “fmt”

b. Print, Printf, Println → print statements

c. Scan, Scanf, Scanln → get input from users

```
fmt.Scanf("%s", &name)                  // pass by reference
fmt.Scanf("%d", &alphabet_count)        // pass by reference
```

d. Error statement that is needed

```
func main(){
    fmt.Println("Hello from Go!")
    var input string
    var err error
    _, err = fmt.Scanln(&input)
    if err != nil {
            fmt.Println("Error - ", err)
    } else{
            fmt.Println("You entered - ", input)
    }
}
```

e. Other external package packages could be imported to be used

f. Multiple packages could be imported using a shorthand syntax

```
import (
        "fmt"
        "time"
)
```

Same as

```
import "fmt"
import "time"
```

3. Variables

a. Go is statically typed

b. Implicitly defined variable – type is inferred by Go compiler

```
var message = "Hello from Go" // message would of type string
```

c. Explicitly defined variable – type is specified explicitly

```
var message string = "Hello from Go"
```

d. Multiple variables could be defined together

```
var x, y int //both x and y are defined as int
```

e. Multiple variables could be defined together and initialized

```
var x, y int = 5, 10
```

Or

```
var (
        name = "Go"
        age = 5
        isGood = true
)
```

f. Within a function varaibles could be defined using a shorthand syntax without using var keyword

g. Shorthand syntax uses := rather than =

- Option 1 – Explicit type declaration

```
func main(){
    message string := "Hello world" // var not used, := is used instead of =
    fmt.Printf (message + "\n")
}
```

- Option 2 – Type inferred

```
func main(){
    message := "Hello world" // var not used, := is used instead of =
    fmt.Printf (message + "\n")
}
```

4. Type declaration

a. Go's type declaration is different from C/C++ and is very similar to Pascal – variable / declared name appears before the type

```
var message = "Hello from Go" // Implicit type declaration – type inferred

var message string = "Hello from Go" // Explicit type declaration
```

5. Constants

a. Const declares a constant value, that can not change

b. A const statement can appear anywhere a var statement can

```
Const PI float32 = 3.14
PI = 3.15 // does not compile and shows a compile time error
```

6. Scope

    a. Go supports package level scoping, function level scoping and block level scoping

    b. Block level Scoping:

```
y := 20
fmt.Println(y)       // 20
{
        y := 10
        fmt.Println(y)       // 10
}
fmt.Println(y)       // 20
```

7. Types

    a. Numbers

        - Integer
            - Signed – int, int16, int32, int64
            - Unsigned – uint8, uint16, uint32, uint64, int8
            - Uint means “unsigned integer” while int means “signed integer”
        - Float
            - Float32, float64

    b. String

    c. Boolean (true, false)

8. Array

    a. Array is a numbered sequence of elements of fixed size

    b. When declaring the array typically the size is specified, though alternately compiler can infer the length

```
var cities[3] string // syntax 1 of declaring arrays
var cities [3]string // syntax 2 of declaring arrays

cities[0] = “Kolkata”
cities[1] = "Chennai"
cities[2] = "Blore"
```

```
fmt.Println(cities[2])        // Blore

cities[2] = "Minneapolis"
fmt.Println(cities[2])        // Minneapolis
```

c. Array size is fixed – it could not be changed after declaring it

d. Arrays could be declared and initialized in the same line

```
// Option 2 - declaring and initialing the array in the same line

cities := [3]string {"Kolkata", "Chennai", "Blore"}

fmt.Println(cities[2])        // Blore

cities[2] = "Minneapolis"
fmt.Println(cities[2])        // Minneapolis
```

e. Go compiler can calculate the length of the array if not specified explicitly

```
cities := [...]string {"Kolkata", "Chennai", "Blore"}
fmt.Println(cities[2])        // Blore
fmt.Println(len(cities))      // 3
```

f. Determining the length of the array

```
//builtin len returns the length of an array
fmt.Printlln(len(cities))     // 2
```

g. Iterating through the array

```
for index, value := range cities {
    fmt.Println(index, value)
}
```

9. Slices

a. Slices are a key data type in Go, giving a more powerful interface to sequences

than arrays. Typically Slices are used in Go rather than array

b. Internally Go uses arrays for slices, but slides are easier to use and more effective

```
cities := []string {"Kolkata", "Bangalore", "Mumbai"}
// slices are declared with syntax similar to array. Only the length is not specified.
// cities := [3]string {"Kolkata", "Bangalore", "Mumbai"}
// Array declaration with length

fmt.Println(cities) // [Kolkata Bangalore Mumbai]
```

```
fmt.Println(cities[1]) // Bangalore

cities = append(cities,"Amsterdam")
cities = append(cities,"Den Haag")

fmt.Println(cities) // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
```

c. Built in len function returns the length of a slice

```
fmt.Println(len(cities)) // 5 is the length of the slice cities
```

d. Slices could also be defined by built-in make function

```
cities := make([]string, 3)
cities[0] = "Kolkata"        // Allows to set values like Arrays
cities[1] = "Bangalore”
cities[2] = "Mumbai"

fmt.Println(cities) // [Kolkata Bangalore Mumbai]
fmt.Println(cities[1]) // Bangalore

cities = append(cities,"Amsterdam")
cities = append(cities,"Den Haag")
fmt.Println(cities) // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
fmt.Println(len(cities)) // 5
```

e. Slices can be also be copied/cloned using copy function

```
fmt.Println(slice) // [Kolkata Bangalore Mumbai Amsterdam Den Haag]

duplicateSlice := make([]string, len(slice))
copy(duplicateSlice, slice)

fmt.Println(duplicateSlice) // [Kolkata Bangalore Mumbai Amsterdam Den Haag]
```

f. Slices support a “slice” operator with the syntax slice [low: high] – same as List processing on other languages

```
fmt.Println(slice[0:2]) // [Kolkata Bangalore]
fmt.Println(slice[:3]) // [Kolkata Bangalore Mumbai]
fmt.Println(slice[2:]) // [Mumbai Amsterdam Den Haag]
```

10. Map

a. Map is one of the built in data structure that Go provides. Similar to hashes or

dicts in other languages

```
employees := map[int]string {
1: "Rob Pike",
2: "Ken Thompson",
3: "Robert Griesemer",
}

fmt.Println(employees) // map[1:Rob Pike 2:Ken Thompson 3:Robert Griesemer]

// Get a value for a key with name[key]
fmt.Println(employees[2]) // Robert Griesemer

// Set a value for a key with name[key]
emps[2] = "Satya Nadela"
fmt.Println(employees[2]) // Satya Nadela
```

b. Maps could be also declared using built in make function

```
// make(map[key-type]val-type)
emps := make(map[int]string)
emps[1] = "Bill"
emps[2] = "Satya"
emps[3] = "Sunder"
emps[4] = "Andrew"

fmt.Println(emps) // map[1:Bill 2:Satya 3:Sunder 4:Andrew]

// Get a value for a key with name[key]
fmt.Println(emps[2]) // Satya

// Set a value for a key with name[key]
 emps[2] = "Satya Nadela"
fmt.Println(emps[2]) // Satya Nadela
```

c. The builtln len returns the number of key/value pairs when called on a map

```
fmt.Println(len(emps))      // 4
```

d. The builtln delete removes key/value pairs from a map

```
delete(emps, 1) // remove element with key 1
delete(emps, 2) // remove element with key 2

fmt.Println(emps) // map[3:Sunder 4:Andrew]
```

11. Error checking

    a. Option 1

```
good, err := some_function()
if err != nil {
    log.Println("function errored")
}

log.Println("we have received from the function successfully {}", good)
```

    b. Option 2

```
if err := second_function(); err != nil {
    log.Println("Error is {}", err)
}
```

    c. Option 3

```
func some_function() (string, int, error){
        return "hello", 2, errors.New("some_function sucks, not working")
}
func second_function error{
        return nil
}
```

12. Pointer

    a. Go uses pointers

    b. Example

```
i := 21
log.Println("value of i is {}", i)
p := &i      //assigning reference to p
*p = 100    // dereferencing p to change the value
log.Println("value of i is {}", i) //value of I has changed
```