CAS CS 350
Lab 2

1. Functions
    a. Functions are declared with funckeyword
    b. Functions can take zero or more arguments and can return values

```
func main (){
    fmt.Println("Hello from Go")
}

func displayMessage(message string){
    fmt.Println(message)
}

displayMessage("Hello")

func add(x int, y int) int {
    return x+y
}

var result int = add(20,10)
fmt.Println(result)
```

    c.
    d. When function arguments are of same type it could be shortened

```
// func add(x int, y int) int {

func add(x, y int) int {
    return x+y
}

var result int = add(20,10)
fmt.Println(result)
```

    e.
    f. Go supports Anonymous functions

Normal Function

```
package main
import "fmt"

func main(){
       var res int = add(20,10)
       fmt.Println(res)     //
       30
}
// add declared as a normal function

func add(x int, y int) int {
       return x+y
}
```

Anonymous Function

```
package main
import "fmt"

    // add declared as an  anonymous function
add := func(x int, y int)  int {
       return x+y
}

fmt.Println(add(20,10))                 // 30

var sub = func(x int, y  int) int {
       return x-y
}

fmt.Println(sub(20,10))                 //
                                        10
```

    g.
    h. Anonymous functions could be called immediately
    i. Very similar to Javascript (and other functional languages)

```
func main(){
        func(msg string){
                fmt.Println("Hello " + msg)
        }("Aniruddha")
}
// Displays Hello Aniruddha

func main(){
        func(num1 int, num2 int){
                fmt.Println(num1 + num2)
        }(10, 20)
}
// Displays 30
```

j.

k. Anonymous functions could be passed as arguments to other functions, and could be returned from other functions. Functions behave like values - could be called function values

l.

```
func main () {
      add := func(x int, y int) int {
            return x+y
      }

      sub := func(x int, y int) int {
            return x-y
      }

      result := doWork(add, 30, 20)
      fmt.Println(result)                    //
                                             1300
      result = doWork(sub, 30, 20)
      fmt.Println(result)                    //
}                                            500

type HigherFunc func(x int, y int) int       // User defined function type

func doWork(anonymous HigherFunc, num1 int, num2 int )
      int{ return anonymous(num1 * num1, num2 * num2)
}
```

m.

n. Variadic functions can be called with any number of trailing arguments

```
func displayMessage(message string, times int,params …string){
    fmt.Println(message, times, params)
}

displayMessage("Call1", 1)
displayMessage("Call2", 2, "Param1")
displayMessage("Call3", 3, "Param1", "Param2", "Param3", "Param4")
```

Output:

```
Call1 1 []
Call2 2 [Param1]
Call3 3 [Param1 Param2 Param3 Param4]
```

o.

2. Control Structures - If
    a. The if statement looks as it does in C or Java, except that the () are gone and {} are required

```
var salary int = 100

if salary < 50 {
    fmt.Println("you are underpaid")
} else if salary >= 50 {
    fmt.Println("you are sufficiently
    paid")
} else {
    fmt.Println("you are overpaid")
}
```

    b.

3. Control Structures - For
    a. Go has only one looping construct, the for loop
    b. The basic for loop looks as it does in C or Java, except that the () are gone (they are not even optional) and the {} are required

```
for ctr := 0; ctr < 10;   ctr++ {
    fmt.Println(ctr)
}
```

    c.
    d. Go does not support while or do while. Same could be achieved using for

```
var ctr int = 0

// same as while
for(ctr < 5) {
     fmt.Println(ctr)
     ctr++
}
```

    e.
    f. As in C or Java, you can leave the pre and post statements empty

```
ctr:=0
for; ctr < 10;{
     ctr +=1
     fmt.Println(ctr)
}
```

    g.
    h. Semicolons could be dropped: C's while is spelled for in Go

```
ctr:=0
for ctr < 10 {                          // behaves in the same way as while ctr < 100 in C or Java
     ctr +=1
     fmt.Println(ctr)
}
```

    i.
    j. Endless or forever loop

```
for {
     // do something – this loop would          never end
}
```

    k.
4. Conditional Statement - Switch
    a. Go's switch statement is more general than C's - expressions need not be constants or even integers

```
city := "Kolkata"

switch city {
case "Kolkata":
       println("Welcome to Kolkata")
      break
case "Bangalore":
       println("Welcome to Bangalore")
      break
case "Mumbai":

       println("Welcome Mumbai")
       break
}
```

```
rating := 2

switch rating {
case 4:
     println("You are rated Excellent")
      break
case 3:
     println("You are rated Good")
      break
case 2:
       println("You are rated Consistent")
       break
case 1:
       println("You need to improve a bit")
       break
}
```

    b.

c. Type switch
   i. used to discover the dynamic type of an interface variable. Such a type switch uses the syntax of a type assertion with the keyword type inside the parentheses.
   ii. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause. It's also idiomatic to reuse the name in such cases, in effect declaring a new variable with the same name but a different type in each case

```
type Human interface
    { Display()
}
type Employee struct {
    name string;
    designation string
}
func (emp Employee) Display() {
    fmt.Println(emp.name, emp.designation)
}
type Contractor struct {
    name string; weeklyHours
    int
}
func (cont Contractor) Display(){
    fmt.Println(cont.name,
    cont.weeklyHours)
}
```

```
func main() {
        var human Human
        human = Employee {
            name:"Aniruddha", designation:"AVP"}

        switch human:= human.(type {
                default:
                        fmt.Println("default")
                case Employee:
                        fmt.Println("Human",
human.designation)
                case Contractor:
                  fmt.Println("Cont", human.weeklyHours)
        }
}
```

iii.

5. Struct
   a. Structs are typed collections of named fields. It is useful for grouping data together to form records
   b. type keyword introduces a new type. It's followed by the name of the type, the keyword struct to indicate that we are defining a struct type and a list of fields inside of curly braces
   c. Go does not have Class, it supports Struct and Interfaces

```
type Employee struct {
    name string            // field name of type string
    age int                // field age of type int
    salary float32         // field salary of type float32
    designation string     // field designation of type
}                                  string
```

   d.
   e. Struct - initialization

Initialization Option 1 – Using new function

```
emp := new(Employee)
emp.name = "Ken Thompson"; emp.age = 50
emp.salary = 12345.678; emp.designation = "Distinguished Engineer"
```

Initialization Option 2 (more like JavaScript)

```
emp := Employee{}
emp.name = "Ken Thompson"; emp.age = 50
emp.salary = 12345.678; emp.designation = "Distinguished Engineer"
```

Initialization Option 3 – parameters should be in the same order fields are declared

```
emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished Engineer"}
fmt.Println(emp)
fmt.Println(emp.name)

// age and salary is not known and so not initialized
newEmp := Employee{name:"New Emp",
designation:"Engineer"}
fmt.Println(newEmp.designation)
```

   f.

g. Structs can have arrays and other child structs as fields

```
type Employee
    struct{ Name
    string
    Age int
    Slills [4]string        // Array field
    HomeAddress Address     // Nested      struc  as propert
    Salary float32             Child       t         y

}
type Address
    struct{ StreetAddress
    string City string
    Country string
}

func main(){
    address := Address{"M G Road", "Bangalore", "IN"}
    skills := [4]string {"C","C++","Go","Rust"}
    emp := Employee{"Aniruddha", 40, 123.456, skills, address}
    fmt.Println(emp.Skills)                    // [C C++ Go Rust]
    fmt.Println(emp)    // {Aniruddha 40 123.456 [C C++ Go Rust] {M G Road Bangalore IN}}
    fmt.Println(emp.HomeAddress.StreetAddress) // MG Road
}
```

h.

6. Go supports methods defined on struct types
    a. Methods of the struct are actually defined outside of the struct declaration

```
type Employee struct {
}

// Method for struct Employee – defined outside of the struct declaration
// Since Display accepts a parameter of type Employee it's considered a member of
Employee
func (emp Employee) Display(){
    fmt.Println("Hello from Employee")
}
func main() {
    emp := Employee{}
    // method
    invokation              // displays "Hello from
    emp.Display()           Employee"
}
```

b.

c. Methods can be defined for either pointer or value receiver types

```
type Employee struct
    { name string
    age int
    salary
    float32
    designation string
}
// Method for struct Employee – this is value receiver type
func (emp Employee) Display() {
    fmt.Println("Name:", emp.name, ", Designation:", emp.designation)
}
func main() {
    emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished
    Engineer"}
    emp.Display()
    // displays Name: Ken Thompson, Designation: Distinguished
    Engineer
}
```

d.

e. Methods can accept parameter similar to functions

```
type Employee struct
    { name string
    age int
    salary
    float32
    designation string
}
// Method for struct Employee – this is value receiver type
func (emp Employee) Display(message string) {
    fmt.Println(message, emp.name, ", Designation:", emp.designation)
}
func main() {
    emp := Employee{"Ken Thompson", 50, 12345.678, "Distinguished
    Engineer"}
    emp.Display("Hello")
    // displays Hello Ken Thompson, Designation: Distinguished
    Engineer
}
```

f.

g. Methods can be defined for either pointer or value receiver types

```
type Employee struct {
    name string; age
    int
}

// Methods for struct Employee – this is pointer receiver type
func (emp* Employee)
      increaseAgeByOne(){ emp.age++
}
func (emp* Employee) increaseAge(increaseBy
      int){ emp.age += increaseBy
}

emp := Employee{"Ken Thompson", 40}
emp.increaseAgeByOne()     // displays Ken Thompson,
emp.display()              41
emp.increaseAge(5)         // displays Ken Thompson,
emp.display()              46
```

h.

```
type Rectangle
    struct{ Height
    float32 Width
    float32
}

// Method that returns a result
func (rect Rectangle) Area() float32{
    return rect.Height * rect.Width
}


rect := Rectangle {Height: 25.5, Width:
12.75} fmt.Println(rect.Area())
```

i.

7. Interface
    a. Interfaces are named collections of method signatures

```
type Human interface { Display()}

type Employee struct {
    name string;
    designation string
}

func (emp Employee) Display(){
    fmt.Println("Name - ", emp.name, ", Designation - ", emp.designation)
}

type Contractor struct {
    name string;
    weeklyHours int
}

func (cont Contractor) Display(){
    fmt.Println("Name - ", cont.name, ", Weekly Hours - ", cont.weeklyHours)
}

func main(){
    var emp Human = Employee{name:"Rob Pike", designation:"Engineer"}
    emp.Display()
    var cont Human = Contractor{name:"XYZ", weeklyHours:35}
    cont.Display()
}
```

    b.

```
func main(){
    var message = "Hello World"
    fmt.Println("Before function call - " + message)
    displayMessage(message)
    fmt.Println("After function call - " + message)
}

func displayMessage(message string){
    fmt.Println("Before update - " + message)
    message = "Hello World from Go"
    fmt.Println("After update - " + message)
}
```

Output

```
Before function call - Hello World
Before update - Hello World
After update - Hello World from Go After
function call - Hello World
```

    c.

```
func main(){
    var message string = "Hello World"
    fmt.Println("Before function call - " + message)
    displayMessagePointer(&message)
    fmt.Println("After function call - " + message)
}

func displayMessagePointer(message *string){
    fmt.Println("Before update - " + *message)
    *message = "Hello World from Go"
    fmt.Println("After update - " + *message)
}
```

Output

```
Before function call - Hello World
Before update - Hello World
After update - Hello World from Go
After function call - Hello World from Go
```

    d.