Examen 2

Pregunta 1:

- a)

  Voy a seleccionar el lenguaje de Go , mi apellido es Gonzalez y empieza con g.

  - 1a) Go tiene varias estructuras de control de flujo entre ellos tenemos :

    - Secuenciación : En Go se indica el flujo de ejecución del código mediante el signo ‘ ; ‘ , es decir, al finalizar cada linea se debe escribir ‘ ; ’ para indicar el final de esta. Sin embargo Go podrá estar basado en C pero es mas amigable o con facilidades para los usuarios, en caso de no colocar ‘ ; ’ se infiere que es un fin de linea.
    - Selección : Go cuenta con condicionales, para poder decidir que código u acción se debe ejecutar bajo ciertas condiciones. Se realiza de mediante las palabras if , else if o else . Adicionalmente también tiene los switch cases.

```
package main

import "fmt"

func main() {
  value:= 2

  if balance < 0 {
    fmt.Println("Value minor than 0")
  } else if balance == 0 {
    fmt.Println("Value is equal to 0")
  } else {
    fmt.Println("Value is greater than 0")
  }

  switch value {
    case value < 0 :
      fmt.Println("It works here too !!")
    case value == 0 :
      fmt.Println("Its cool right?")
    default :
      fmt.Println("default case !@#@#! !!")
  }
}
```

- Repetición: tenemos también los ciclos, en go se deben expresar con for (No) de la siguiente manera :

```
package main

import "fmt"

func main() {
  var n,value = 5 , 0
  for i:=1 ; i<=n ; i++ {
    value += i
  }

  fmt.Println("actual value =",value)
  /*
  Si necesitamos un while loop , debemos hacerlo con for
  de la siguiente manera
  */

  for value > 0 {
    fmtPrintln("Decreased value", value)
    value = value - 1
  }

}
```

- Abstracción procedural : nos permite reciclar nuestro código mediante funciones. Se crean mediante la palabra func más un nombre y los parámetros de la función.

```
package main

import "fmt"

func helloNewWorld ( world int ) int {
  fmt.Println("Hello  world %d",world)
  return 0 // all went well
}

func main() {
  var n,value = 5 , 0
  for i:=1 ; i<=n ; i++ {
    helloNewWorld(i)
  }
}

/*
```

```
  Imprimira el string n veces
*/
```

- Recursion: Go es capaz de realizar recursiones, simplemente creamos una función que se llame a si misma.

```
package main

import "fmt"

func fact ( n int ) int {
  if n == 0 {
   return 1
  }
  return n * fact(n-1)
}

func main() {
  fmt.Println(fact(3))

/*
  Otra manera de declarar una recursion es
*/

  var fib func(n int ) int

  fib = func(n int) int {
    if n < 2 {
      return n
    }

    return fib(n-1) + fib (n-2)
  }

  fmt.Println(fib(7))
}
```

- Concurrencia: Existen las goroutines y channels (se complementan entre si), adicionalmente existen funciones que realizan operaciones sobre ellos para poder tener implementaciones más robustas.

```
package main

import "fmt"
```

```
func hi ( n string )  {
  for i:=0 ; i < 2 ; i++ {
    fmt.Println(n,"  ", i)
  }
}

func main() {

  // llamada normal
  hi("normal")

  // llamada concurrente utilizando goroutines

  go hi ("Concurrente")
}
```

- Finalmente excepciones : Go nos ofrece control de errores a traves de una interfaz llamada error. Con esta interfaz de error podemos crear mensaje para el usuario, verificar si ocurrieron o no errores, etc.

```
package main

import "fmt"
import "errors"


func main() {

  var x int = 2
  if x > 3 {
    return errors.New("FALLO TERMINAL")
  }
}
```

- 1b) En go las expresiones son evaluadas de izquierda a derecha, es decir, de manera infija, esto implica que se puede indicar el orden de evaluación de ciertas expresiones con paréntesis (de manera explicita) o con reglas de precedencia y asociatividad (de manera implícita). Adicionalmente al evaluar las expresiones booleanas de la forma a && b se comienza desde la izquierda, es decir, ‘ a ’ y ‘ b ’ sola sera evaluada si ‘ a ’ ha dado como resultado true. De manera similar en una expresión de la forma a | | b,

sera evaluada ‘ a ’ primero y se evaluara ‘ b ’ solo si ‘ a ’ ha resultado en false. Lo anterior es una especie de cortocircuito.

Como información adicional las expresiones (y esto incluye a las funciones) son evaluadas luego de que han sido evaluadas las expresiones de las que estas dependen, es decir, si tenemos una función hello(x , g(c)) para poder evaluar dicha función, primero se ha de evaluar hello , x y g(c), nuevamente al evaluar g(c) primera sera revisado g y c.

Un par de últimos datos curiosos. En go no existe el operador ternario por decision de desarrollo del lenguaje, básicamente decidieron mantener únicamente los bloques if-else (en caso de ternario) ya que es un código “más claro” al leerlo. Y las expresiones i++ o i-- NO son expresiones son Declaraciones.

- 1c) En Go tenemos datos de tipo simple (o básicos que es como lo catalogan en go) y de tipo compuesto.

  Los de tipo simple son : string , bool y un catalogo extenso de tipo numérico que incluye desde int8,float32 a complex128, etc.

  Los de tipo compuesto : pointer y struct (similares a los de C), funciones (que son de primera clase en Go), tipo contenedor que incluye a arrays, slice y map, tipo channel (que son usados para sincronizar data entre las goroutines) y tipo interface que son utilizados para polimorfismos y reflexiones.

  Adicionalmente Go facilita la creación de nuevos tipos a partir de la palabra ‘type’ , el nombre del tipo a crear y una fuente del tipo.

```
/*
  De manera que stringByUs es en realidad un int

*/
type stringByUs int


/*
  Se pueden declarar nuevos tipos en simultaneo.

*/
type (
  hiL = string
  century = int
)

/*
  Incluso se pueden crear nuevos tipos a partir
```

```
  de tipos creados por el prograador.
*/

type (
  ourFloat float32
  totalPayment ourFloat
)
```

- 1d) En Go estan presente la equivalencia por nombres y la estructural. En el caso de al equivalencia por nombres, tenemos que los tipos definidos son diferentes de otro tipo. Y en el caso de equivalencia estructural, dos tipos son iguales si los subtipos internos tienen la misma estructura y no son de un tipo con nombre.

```
type A struct {
  value int32
}

type B struct {
  value int32
}

/*
    Typo A es identico a tipo B.
    En este caso A.value y B.value son iguales y por lo tanto A == B.
    Cuando se hace la comparcion de A.value y B.value se verifica que
    tienen el mismo nombre.
*/
```

Cuando se declaran variables en Go, no siempre es necesario expresar de manera explicita el tipo, Go es capaz de inferirlo.

```
/*
  En este caso value ha sido declarado como un tipo int
  y sera inferido que el tipo de cost es int
*/
var value int
cost := value

/*
  Tambien se pueden crear variables directamente con valores
  sin declaracion de tipos
*/

age := 23 // int
capacity := 798.565 // float
```

Finalmente go al ser un lenguaje compilado, verifica la compatibilidad de los datos y variables al ser compilado, de manera que si hay un error de tipos o incompatibilidad de los mismo, la compilacion fallara.