Modelo de Examen – Lenguaje Go (Golang)

Instrucciones generales:

- Responde las preguntas de forma clara y completa.
- Cuando se indique, escribí el código necesario de forma correcta y funcional.
- Justifica tus respuestas cuando corresponda.

◆ Parte I – Teoría conceptual (preguntas a desarrollar)

1. **Lenguaje y usos generales**
    - ¿Qué tipo de lenguaje es Go (características principales) y en qué áreas suele utilizarse?
    - ¿Por qué es recomendado para desarrollo concurrente y en la nube?

    Rta 1: Go es un lenguaje compilado y fuertemente tipado y suele utilizarse en el área del desarrollo web en la parte del back-end y en aplicaciones concurrentes.
    Es recomendado para el desarrollo concurrente porque go soporta la concurrencia y es rápido en tiempo de ejecución y compilación.

2. **Estructura de un programa en Go**
    - ¿Qué es un paquete en Go y por qué todo programa debe comenzar en `package main`?
    - Explica el comportamiento de los identificadores que comienzan con mayúscula vs minúscula.

    Rta 2: Un paquete en go es una forma de organizar el código y definir su estructura, todo programa debe pertenecer al paquete main, para ser ejecutable, porque la ejecución de un programa empieza en la función main().
    Cuando un identificador empieza con mayúscula hace que el mismo pueda ser "exportado", es decir que sea visible desde afuera (osea público), y al contrario cuando es con minúscula (privado).

3. **Declaración de variables y constantes**
    - ¿Cuál es la diferencia entre declarar una variable con `var` y con:=?
    - ¿Qué son los "zero values" y por qué son relevantes en Go?

    Rta 3: cuando se declara una variable con la palabra "`var`" se tiene que poner explícitamente el tipo de la variable, y no es necesario inicializar, en cambio cuando se usa "**:**=" se infiere el tipo de la variable y a la vez se inicializa. Los zero values son valores iniciales que Go asigna a las variables cuando son declaradas, pero no inicializadas, son relevantes porque se

asegura que las variables tengan un valor inicial conocido y para no tener que inicializar explícitamente cada variable.

4. **Pasaje de parámetros**
   - Explica qué significa que Go pasa parámetros “por valor”, pero que aun así puede modificarse una variable externa mediante un puntero.
   - Ilustra tu explicación con un pequeño ejemplo de función que modifique el valor original de una variable usando punteros.

   Rta 4: En Go los parámetros se pasan por valor, osea que se pasa una copia, por lo que cualquier modificación realizada no va a afectar a la variable original fuera de la función, pero se puede modificar una variable externa mediante un puntero, pasando la dirección del mismo (usando &), la función puede desreferenciar (usando *) esa dirección de memoria, para acceder y modificar en memoria al mismo, de esta manera, aunque el puntero se pase por valor, su contenido si puede ser modificado.

```
num := 10

fmt.Println("-----Variable Inicializada-----")
fmt.Println(num) //Imprime 10

multiplicarX2(num)

fmt.Println("-----Variable Pasada por valor-----")
fmt.Println(num) //Imprime 10

multiplicarX2Refe(&num)

fmt.Println("-----Variable Pasada por 'Referencia'-----")
fmt.Println(num) //Imprime 20
```

5. **Funciones con retorno nombrado**
   - ¿Qué son los parámetros de retorno nombrados y cómo afectan la forma de retornar valores?
   - Escribí una función que reciba dos enteros y retorne su suma y su resta usando parámetros de retorno nombrados.

   Rta 5: Los parámetros de retorno nombrados en Go son variables que se declaran directamente en la firma de una función, su uso hace que se pueda hacer el retorno implícito, que es usar el return sin poner argumentos, y Go automáticamente retorna los valores actuales de las variables de retorno nombradas.

```go
func calcular(a, b int) (suma, resta int) { // Declaración de parámetros de retorno nombrados
    // Asigna valores a los parámetros de retorno nombrados directamente
    suma = a + b
    resta = a - b
    // Retorno implícito: al usar 'return' sin argumentos, Go retorna los valores actuales de las variables de retorno nombradas (suma y resta en este caso)
    return
}
```

6. **Estructuras de control**
    - ¿Qué diferencias hay entre un `switch` con selector y uno sin selector en Go?
    - Proporciona un ejemplo en el que tenga más sentido usar un `switch` sin selector que uno con selector.

    Rta 6: El switch con selector incluye una variable o expresión (el "selector") inmediatamente después de la palabra "switch", el switch común, se comporta parecido a una cadena de if-else, cada case debe tener una condición y el switch ejecuta la primera que sea verdadera

```go
    t := time.Now() // Obtenemos la hora actual

    // Este switch no tiene un selector explícito después de la palabra clave 'switch.
    // Cada 'case' evalúa una expresión booleana independiente.
    switch {
    case t.Hour() < 12: // Si la hora es menor a 12 (antes del mediodía)
        fmt.Println("¡Buenos días!")
    case t.Hour() < 17: // Si la hora es menor a 17 (antes de las 5 PM)
        fmt.Println("¡Buenas tardes!")
    default: // Si ninguna de las condiciones anteriores es verdadera
        fmt.Println("¡Buenas noches!")
    }
```

7. **Colecciones**
    - Explica las diferencias entre arrays y slices en Go. ¿Por qué se dice que los slices son más flexibles?
    - ¿Qué pasa si se modifica un slice? ¿Afecta al array subyacente? Justifica con un ejemplo simple.

    Rta 7: Los arrays son una estructura de datos utilizada para almacenar una secuencia de elementos con una longitud fija, osea la capacidad de elementos máxima. En cambio, los slices son una estructura de datos similar a los

arrays, pero son un segmento de un array subyacente, son como arrays dinámicos en la que la capacidad de elementos puede cambiar y son flexibles debido a esto mismo. Cuando se modifica un slice esa modificación afecta a el array subyacente, porque los slices son una referencia a ese array subyacente, por lo que también a los slices que referencien al mismo.

```go
// Creo un array de 5 elementos
arr := [5]int{1, 2, 3, 4, 5}

// Creo un slice que apunta a los primeros 3 elementos del array
slice := arr[0:3] // slice contiene [1 2 3]

// Modifico el segundo elemento del slice
slice[1] = 99

// Imprimo el slice y el array original
fmt.Println("Slice:", slice) // [1 99 3]
fmt.Println("Array:", arr)   // [1 99 3 4 5] ← ¡se modificó!
```

8. **Maps en Go**
   - ¿Qué condiciones deben cumplir las claves de un `map` en Go? ¿Qué sucede si se intenta usar un tipo no permitido?
   - ¿Cómo se verifica si una clave existe en un map antes de acceder a su valor?

   Rta 8: Las claves de un Map en go tienen que ser tipos que soporten la comparación por igualdad (==) osea que sean comparables, si se intenta usar un tipo no permitido da error en tiempo de compilación.
   Para verificar si una clave existe se usa la sintaxis:
   `valor, ok := miMap[clave]`
   valor es la variable donde se almacena el valor asociado a la clave, si la clave no existe valor tomara el zero value correspondiente al tipo, **ok** es una variable booleana que indica si la clave existía o no en el map.

9. **Concurrencia**
   - ¿Qué son las Goroutines? ¿Qué ventajas tienen respecto a los hilos tradicionales?
   - Explica brevemente cómo funciona un `WaitGroup` para esperar la finalización de varias Goroutines concurrentes.

   Rta 9: Las Goroutines son funciones que tienen la capacidad de ejecutarse concurrentemente con otras funciones, la ventaja que tienen es que son más ligeras, eficientes y generan alta escalabilidad. Un WaitGroup es una forma de esperar a que varias Goroutines terminen su ejecución a través de sus métodos Add(n), que aumenta en n el “contador”, Done() que decrementa en uno el contador y Wait() que bloquea la ejecución hasta que el “contador” sea 0.

10. **Channels**
    - ¿Qué son los Channels y cómo permiten la comunicación entre Goroutines?
    - ¿Qué diferencia hay entre un channel sin búfer y uno con búfer?

    Rta 10: Los Channels o canales son un mecanismo que permite que las Goroutines se comuniquen y se sincronicen, permiten la comunicación entre Goroutines a través del envio y la recepción de mensajes.
    Un canal sin búfer cuando envía se bloquea hasta que se reciba y viceversa, en cambio el canal con búfer puede almacenar varios valores sin que alguien los reciba inmediatamente, el envio solo se bloque si el búfer está lleno y la recepción se bloquea si esta vacío.

◆ Parte II – Preguntas con código

11. **Slices dinámico**
    - Escribí un programa que cree un slice de enteros, le agregue elementos con `append()`, y luego lo recorra con `for range`, imprimiendo cada elemento.

```
    sliceEnt := []int{} //Creo el slice de enteros vacío

    sliceEnt = append(sliceEnt, 10, 20, 30) //Agrego elementos con append

    for i, valor := range sliceEnt {    //Recorro con for range
        fmt.Println("Elemento en posición", i, "=", valor)
    }
```

12. **Punteros**
    - Escribí una función que reciba un puntero a `int` y lo incremente en 1. Mostrá su uso en `main`.

```
func incrementar(num *int) {
    *num = *num + 1
}
    //Main
    num = 10
    incrementar(&num)
    fmt.Println(num) //Imprime 11
```

13. **Maps**
    - Escribí un programa que cree un `map[string]int` para almacenar la cantidad de veces que aparece cada palabra en un slice de strings.

```
    m := make(map[string]int)
    slicePalabras := []string{"Perro", "Caballo", "Conejo", "Perro",
"Cebra"}

    for _, valor := range slicePalabras {
        m[valor] = m[valor] + 1
    }

    fmt.Println(m)
```

14. **Select y Channels**
    - Escribí un pequeño ejemplo que usé dos canales y un `select` para recibir un valor de cualquiera de ellos.

```
    ch1 := make(chan string)
    ch2 := make(chan string)

    go func1(ch1) // Goroutine que envía un mensaje después de 1 segundo
    go func2(ch2) // Goroutine que envía un mensaje después de 2 segundos

    select {
    case msg1 := <-ch1:
        fmt.Println("Mensaje recibido del canal 1", msg1)
    case msg2 := <-ch2:
        fmt.Println("Mensaje recibido del canal 2", msg2)
    }
```

◆ Parte III – Pregunta de comprensión avanzada (opcional)

15. **Genéricos**

- Explica qué problema resuelven los genéricos en Go y cómo se definen.
- Escribí una función genérica que calculé la suma de los valores en un `map[K]V`, donde `V` puede ser `int64` o `float64`.

Rta 15: Los genéricos en Go resuelven el problema de no tener que escribir lo mismo para varios tipos de datos y poder hacer función o algo genérico para **no repetir el mismo código (solo cambiando el tipo de dato) varias veces**,
Se definen con **parámetros de tipo entre corchetes** `[]`.

```go
func SumValues[K comparable, V int64 | float64](m map[K]V) V {
    var s V // Declara una variable 's' del tipo V para almacenar la
suma,
    // inicializada con el zero value de V (0 para int64/float64)
    for _, v := range m { // Itera sobre los valores del mapa
        s += v // Suma cada valor a 's'
    }
    return s // Retorna la suma total
}
```