MODELO DE EXAMEN: LENGUAJE GO (GOLANG)

Instrucciones generales:

• Lee cada pregunta cuidadosamente.
• Responde de manera clara y concisa.
• Para las preguntas de código, escribe el código Go **válido y funcional**.
• Justifica tus respuestas cuando corresponda, citando las fuentes pertinentes.

Parte I – Teoría Conceptual (Preguntas a desarrollar)

1. Fundamentos de Go:

◦ **Define qué es Go y menciona al menos tres de sus principales usos.**

▪ Go, a menudo referido como "Golang", es un lenguaje de programación multiplataforma de código abierto, desarrollado por Google en 2007. Es un lenguaje compilado y fuertemente tipado, con una sintaxis basada en C/C++.
▪ Se utiliza principalmente para desarrollo web (lado del servidor/back-end), desarrollo de aplicaciones en red, aplicaciones concurrentes, desarrollo de aplicaciones multiplataforma y desarrollo nativo en la nube.

◦ **Explica dos razones por las cuales se recomienda usar Go.**

▪ Go es fácil de aprender y portable a diferentes plataformas (Windows, Mac, Linux). También es valorado por su rapidez en tiempo de ejecución y compilación, soporte para concurrencia y genéricos, y gestión eficiente de la memoria.

2. Estructura y Visibilidad:

◦ **¿A qué `package` debe pertenecer todo programa ejecutable en Go y por qué?**

▪ Todo programa ejecutable debe pertenecer al paquete `main`. La ejecución de un programa Go comienza en la función `main()` dentro de este paquete.

◦ **Explica la diferencia de comportamiento de los identificadores (variables, funciones, etc.) que comienzan con mayúscula versus los que comienzan con minúscula.**

▪ Los identificadores que comienzan con **mayúscula** son "exportados", lo que significa que son visibles y accesibles desde otros paquetes (son públicos). Por el contrario, los identificadores que comienzan con **minúscula** son privados y solo son visibles dentro de su propio paquete.

3. Variables y Constantes:

◦ **¿Cuál es la diferencia entre declarar una variable con la palabra clave `var` y con la sintaxis corta `:=`? ¿Cuándo se puede usar `:=`?.**

▪ Cuando se declara una variable con `var`, se puede incluir explícitamente el tipo de la variable (ej. `var a int = 10`) y no es necesario inicializarla inmediatamente. Go le asignará su "**zero value**" por defecto si no se inicializa.
▪ La sintaxis corta `:=` (ej. `b := 20`) se utiliza para la **declaración y asignación simultánea**, donde Go infiere automáticamente el tipo de la variable a partir del valor de inicialización. Esta sintaxis **solo se puede usar dentro de funciones**.

◦ **Menciona el valor por defecto (zero value) para los siguientes tipos básicos en Go: `bool`, `string`, `int`, `float64`.**

- ▪ `bool`: **false**.
- ▪ `string`: "" (cadena vacía).
- ▪ `int`: **0**.
- ▪ `float64`: **0**.

▪ Los "**zero values**" son relevantes porque aseguran que las variables tengan un valor inicial conocido, evitando la necesidad de inicializar explícitamente cada variable.

4. Pasaje de Parámetros y Punteros:

◦ **Go utiliza un modelo de pasaje de parámetros "por valor". Explica qué significa esto y cómo, a pesar de ello, se puede modificar un valor externo a la función utilizando punteros.**

▪ "Pasaje por valor" significa que cuando se llama a una función, se crea una **copia de cada argumento** y esa copia se pasa a la función. La función opera sobre estas copias, por lo que las modificaciones no afectan a las variables originales fuera de la función.
▪ Para modificar un valor externo, se deben pasar **punteros** a esos valores como parámetros. Aunque el puntero en sí es una copia (el valor de la dirección de memoria es copiado), el contenido al que apunta puede ser **desreferenciado** (`*`) y modificado dentro de la función, afectando así el valor original en memoria.
Que una variable sea dereferenciada significa que se usa el operador *, para obtener el valor al que apunta, osea, el valor que hay en esa dirección de memoria almacenada en el puntero.

5. Funciones y Retornos Nombrados:

◦ **¿Qué son los parámetros de retorno nombrados en Go y cómo afectan la forma de retornar valores?**

▪ Los parámetros de retorno nombrados permiten **declarar las variables de retorno**

directamente en la firma de la función. Dentro del cuerpo de la función, se pueden asignar valores a estas variables.
▪ Su uso permite un **retorno implícito**: al usar `return` sin argumentos, Go automáticamente retorna los valores actuales asignados a estas variables de retorno nombradas.

6. Estructuras de Control - `switch`:

◦ **Explica el `switch` sin selector en Go. ¿Para qué tipo de situaciones es útil?.**

▪ Un `switch` sin selector en Go funciona como una **cadena de sentencias `if-else if-else`.** La sentencia `switch` ejecuta el **primer `case` cuya expresión es verdadera**.
▪ Es útil para **condiciones complejas** que no se basan en un único valor de una variable, sino en **múltiples expresiones booleanas**.
▪ **No necesita `break`** al final de cada `case`, ya que solo ejecuta el bloque coincidente y sale.

7. Colecciones - Arrays vs. Slices:

◦ **Describe la diferencia fundamental entre un `Array` y un `Slice` en Go, en términos de su longitud y flexibilidad. ¿Por qué se dice que los Slices son más flexibles?**

▪ Un **`Array`** es una secuencia indexada de elementos del mismo tipo, con una **longitud fija** que es parte de su definición de tipo. Su tamaño no puede cambiar una vez declarado.
▪ Un **`Slice`** es un "segmento" de un Array subyacente y son más flexibles que los Arrays. Tienen una longitud (elementos actuales) y una capacidad (cuánto puede crecer sin reasignar memoria), y su **longitud puede cambiar dinámicamente**. Esta capacidad de cambiar su tamaño es lo que los hace más flexibles.

◦ **¿Qué sucede si se modifica un `Slice`? ¿Afecta al `Array` subyacente y a otros `Slices` que lo referencien? Justifica con un ejemplo simple.**

▪ Sí, **cuando se modifica un `Slice`, esa modificación afecta al `Array` subyacente** y, por ende, a cualquier otro **`Slice`** que referencie la misma porción de ese **`Array`**. Esto se debe a que los **`Slices`** son **referencias** a los **`Arrays`** subyacentes.

8. Maps en Go:

◦ **¿Qué condiciones deben cumplir las claves de un `map` en Go? ¿Qué tipos de datos no están permitidos como claves?**

▪ Las claves de un `map` en Go deben ser de tipos que soporten la **comparación por igualdad (`==`)**, es decir, deben ser **comparables**. Esto incluye tipos como **`booleans`**, **`números`** (enteros, flotantes), **`strings`**, y **`arrays`**.
▪ Los tipos que **no están permitidos** como claves son **`slices`**, **`maps`** y **`functions`**. Intentar usar un tipo no permitido como clave dará un error en tiempo de compilación.

◦ **¿Cómo se verifica si una clave existe en un `map` antes de acceder a su valor?.**

▪ Para verificar si una clave existe, se utiliza la sintaxis especial con **dos valores de retorno**: `elem, ok = m[key]` contendrá el valor asociado a la clave (o su "zero value" si la clave no existe), y `ok` será un booleano (`true` si la clave existe, `false` si no).

9. Concurrencia - Goroutines y WaitGroup:

◦ **¿Qué son las Goroutines en Go? ¿Qué ventajas tienen respecto a los hilos tradicionales?**

▪ Las Goroutines son funciones que se ejecutan **concurrentemente** con otras funciones. Son **ligeras y eficientes**, lo que permite lanzar miles o millones de ellas sin incurrir en una gran sobrecarga, facilitando alta escalabilidad.

◦ **Explica brevemente cómo funciona un `sync.WaitGroup` para esperar la finalización de varias Goroutines concurrentes.**

▪ `sync.WaitGroup` es una forma de esperar a que varias Goroutines terminen su ejecución. Funciona como un **contador**.
▪ Sus métodos principales son:

- `Add(n int)`: Incrementa el contador. Se llama antes de lanzar cada Goroutine.
- `Done()`: Decrementa el contador en 1. Se suele usar con `defer` dentro de la Goroutine para asegurar que se decrementa al finalizar.
- `Wait()`: Bloquea la Goroutine que la ejecuta hasta que el contador llegue a cero. Se llama en la función principal para esperar que todas las Goroutines finalicen.

10. Concurrencia - Channels y Select:

◦ **¿Qué son los Channels en Go y cuál es su principal función? ¿Qué diferencia hay entre un channel sin búfer y uno con búfer?**

▪ Los Channels son el **mecanismo principal para la comunicación y sincronización entre Goroutines**. Son conductos "tipados" a través de los cuales una Goroutine envía datos a otra.
▪ **Channel sin búfer (por defecto)**: Las operaciones de envío y recepción son **bloqueantes**. Esto significa que una Goroutine que envía se bloquea hasta que otra Goroutine esté lista para recibir, y viceversa.
▪ **Channel con búfer**: Puede almacenar varios valores sin que sean recibidos inmediatamente. Las operaciones de envío solo se bloquean si el búfer está lleno, y las operaciones de recepción se bloquean si el búfer está vacío.

◦ **¿Para qué se utiliza la sentencia `select` en Go?**

▪ `select` permite que una Goroutine **espere por más de un `channel`**, ya sea para operaciones de envío o recepción. Ejecuta el `case` cuya operación de `channel` esté lista primero.

11. Panic y Recover:

◦ **¿Qué es un `panic` en Go y cuándo se utiliza?**

▪ Un `panic` ocurre cuando Go detecta un **error en tiempo de ejecución**, deteniendo la ejecución normal del programa. También puede ser generado explícitamente con la función `panic()`. Se utiliza típicamente para errores "**no esperables**".

◦ **¿Cuál es el propósito de la función `recover()` y cómo se relaciona con `defer`?.**

▪ `recover()` es una función que permite **recuperar la ejecución ante un `panic`** o al menos dejar el estado prolijo. Solo tiene efecto si es invocada **dentro de una función diferida (`defer`)** durante un `panic`. Cuando un `panic` ocurre, las funciones diferidas se ejecutan, y si `recover()` es llamada en una de ellas, finaliza el estado de pánico y retorna el valor del `panic`. La función que entró en pánico no continúa, pero el programa puede evitar una falla total.

12. Genéricos:

◦ **¿Cuál es el principal problema que resuelven los genéricos en Go?**

▪ Los genéricos resuelven el problema de tener que **escribir funciones duplicadas para la misma operación sobre diferentes tipos de datos**. Permiten escribir código reutilizable y más limpio sin perder seguridad de tipos.

◦ **Menciona y describe tres "type constraints" comunes utilizados en Go para genéricos.**

▪ `any`: Representa cualquier tipo de dato.
▪ `comparable`: Representa tipos que tienen definida la comparación por igualdad (`==`).
▪ **Uniones de tipos específicos** (ej. `int | float64`): Restringen el parámetro de tipo a uno de los tipos listados.

Parte II – Preguntas con Código

1. Slices Dinámicos:

◦ **Escribe un programa Go que:**
- **Cree un `Slice` de enteros vacío.**
- **Le agregue al menos tres elementos utilizando `append()`.**
- **Luego, lo recorra con `for range`, imprimiendo cada elemento junto con su índice.**

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


    sliceEnteros = append(sliceEnteros, 10, 20, 30) //Agrego 3 Elementos

    for i, valor := range sliceEnteros {    //Imprimo con for range
        fmt.Println("Posicion", i, "Valor:", valor)
    }
```

2. Punteros en Funciones:

◦ **Escribe una función en Go que reciba un puntero a un `int` y lo incremente en 1. Muestra su uso en la función `main` para verificar que el valor original de la variable externa se ha modificado.**

```
func incrementar(num *int) {

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

3. Maps para Conteo de Palabras:

◦ **Escribe un programa Go que cree un `map[string]int` para almacenar la cantidad de veces que aparece cada palabra en un `slice` de `strings` dado.**

```
    //Creo el map de clave string, valor int
    m := make(map[string]int)
    //Slice de palabras
    slicePalabras := []string{"Perro", "Caballo", "Conejo", "Perro",
"Cebra"}

    //Recorro el map con for range
    for _, valor := range slicePalabras {
        m[valor] = m[valor] + 1
    }

    fmt.Println(m)
```

4. Concurrencia con `select` y Channels:

◦ **Escribe un pequeño ejemplo que use dos `channels` y la sentencia `select` para recibir un valor de cualquiera de ellos. Simula que un `channel` puede enviar un mensaje antes que el otro.**

```
    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)
    }
```