MODELO EXAMEN TEORICO – SEMINARIO DE LENGUAJES: OPCION ‘GO’

Sección 1: Concurrencia (Goroutines, Channels, Select, Exclusión Mutua)

Pregunta 1.1: Goroutines a) ¿Qué es una "goroutine" en Go y cómo se lanza?

a) Una **goroutine** es una función que es capaz de ejecutarse concurrentemente con otras funciones. Se lanza una goroutine añadiendo la palabra clave `go` antes de la llamada a la función. Por ejemplo: `go f(0).`

Pregunta 1.2: WaitGroup a) Explique el propósito del tipo `sync.WaitGroup` y cuáles son sus tres métodos principales.

a) El tipo `sync.WaitGroup` es una forma de esperar a que varias Goroutines terminen su ejecución. Se puede conceptualizar como un contador. Sus métodos son:
* `Add(n int):` Incrementa (o decrementa) el contador. * `Done():` Decrementa en 1 el contador. * `Wait():` Bloquea a la goroutine que la ejecuta hasta que el contador llegue a cero.

b) Proporcione un breve ejemplo de cómo se utilizarían estos métodos para esperar la finalización de múltiples Goroutines.

b) Para usar `WaitGroup`, se añade `wg.Add(1)` antes de lanzar cada goroutine, y `defer wg.Done()` dentro de la goroutine para asegurar que el contador se decremente cuando la goroutine termine. Finalmente, `wg.Wait()` en la función principal bloquea hasta que todas las Goroutines añadidas hayan completado su ejecución.

```
var wg sync.WaitGroup // Declaración global o pasada por referencia
for _, url := range urls {
    wg.Add(1) // Incrementar el contador por cada goroutine
    go func(url string) {
        defer wg.Done() // Decrementar el contador al finalizar
        responseSize(url) // Función a ejecutar concurrentemente
    }(url)
}
wg.Wait() // Bloquear hasta que el contador sea cero
```

Pregunta 1.3: Channels a) ¿Qué son los "Channels" en Go y cuál es su principal función?

a) Los **Channels** son un mecanismo que permite que las Goroutines se comuniquen y se sincronicen. Son un conducto "tipado" a través del cual una goroutine envía datos a otra.

b) Describa el comportamiento por defecto de las operaciones de envío (`send`) y recepción (`receive`) en un channel no buferizado.

b) Por defecto, tanto la acción de enviar como la de recibir **bloquean** a la goroutine que la ejecuta hasta que la del "otro extremo" (la goroutine que envía o recibe) esté lista.

c) ¿Cómo se declara un channel y cuál es su "zero value"?

c) Un channel se declara usando `make(chan Tipo)` o `var nombre chan Tipo = make(chan Tipo)`. El "zero value" de un channel es `nil`.

d) Explique brevemente la diferencia entre un channel "send-only" y un channel "receive-only". ¿Quién puede cerrar cada tipo?

d) * Un **send-only channel** se declara como `chan <- int` (ejemplo con `int`). Solo la goroutine "sender" (la que envía datos) puede cerrar un send-only channel.
* Un **receive-only channel** se declara como `<-chan int` (ejemplo con `int`). Intentar cerrar un receive-only channel produce un error en tiempo de compilación.

Pregunta 1.4: Select a) ¿Para qué se utiliza la sentencia `Select` en Go?

a) `Select` permite que una goroutine espere por más de un channel, ya sea para operaciones de envío (`send`) o recepción (`receive`).

b) Explique la funcionalidad de la alternativa `default` dentro de un `select`.

b) `select` puede utilizar una alternativa `default` para realizar operaciones de envío o recepción **sin bloqueo**. Si ninguna de las otras operaciones de los `case` está lista para ejecutarse, el bloque `default` se ejecutará inmediatamente.

Pregunta 1.5: Exclusión Mutua (Mutex) a) Describa el problema de la exclusión mutua en la concurrencia y cómo lo aborda `sync.Mutex` en Go.

a) El problema de la exclusión mutua surge cuando múltiples Goroutines intentan acceder y modificar simultáneamente una misma variable compartida, lo que puede llevar a resultados inesperados y erróneos. `sync.Mutex` aborda esto bloqueando el acceso a una sección crítica de código, asegurando que solo una goroutine pueda ejecutar esa sección a la vez.

b) ¿Cuáles son los dos métodos principales del tipo `sync.Mutex` y cuál es su comportamiento?

b) Los dos métodos principales de `sync.Mutex` son: * `Lock()`: Bloquea el Mutex. Si el Mutex ya está bloqueado, la goroutine que invoca a `Lock()` se bloquea hasta que otra goroutine invoque a `Unlock()`. * `Unlock()`: Desbloquea el Mutex. Si el Mutex no está bloqueado, se produce un error en tiempo de ejecución.

c) ¿Cuál es la diferencia entre `sync.Mutex` y `sync.RWMutex`?

c) `sync.Mutex` es un semáforo binario que proporciona exclusión mutua para escritura y lectura (`Lock()` bloquea el acceso de todas las demás Goroutines).
`sync.RWMutex` es un semáforo "**un escritor – múltiples lectores**". Permite que múltiples lectores accedan a un recurso compartido concurrentemente (`RLock()`), pero solo permite que un único escritor acceda exclusivamente al recurso (`Lock()`). Si hay lectores activos, `Lock()` se bloquea hasta que todos los lectores hayan terminado (`RUnlock()`).

Pregunta 1.6: Panic y Recover a) ¿Qué es un `panic` en Go y cuándo se utiliza?

a) Un `panic` ocurre cuando Go detecta un error en tiempo de ejecución, deteniendo la ejecución normal del programa. También puede ser generado explícitamente por el programa mediante la función `panic()`. Se utiliza típicamente para errores "no esperables", mientras que para errores "esperables" es mejor usar el mecanismo de retorno de error.

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

b) `recover()` es una función que permite recuperar la ejecución ante la ocurrencia de un `panic`, o al menos dejar el estado prolijo. `recover()` solo tiene efecto si es invocada dentro de una **función diferida (`defer`)**. Cuando una función entra en pánico, las funciones diferidas se ejecutan, y si `recover()` es llamada dentro de 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. Si `recover()` es invocada en cualquier otro momento (fuera de un `defer` durante un panic), no tiene ningún efecto y retorna `nil`.

Sección 2: Genéricos

Pregunta 2.1: Introducción a Genéricos a) ¿Cuál es el principal problema que resuelven las funciones genéricas en Go?

a) Las funciones genéricas resuelven el problema de tener que escribir funciones duplicadas para realizar la misma operación sobre diferentes tipos de datos.

b) Explique la sintaxis básica para declarar una función genérica en Go, incluyendo los "type formal parameters" y "type constraints".

b) Una función genérica se declara con la palabra clave `func` seguida del nombre de la función, y luego una lista de **"type formal parameters"** entre corchetes `[]` antes de los parámetros regulares. Cada parámetro de tipo puede tener un **"type constraint"** que define qué tipos de datos pueden reemplazar ese parámetro de tipo. Por ejemplo: `func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V`. Aquí, `K` es un parámetro de tipo con la restricción `comparable` (**es decir, el tipo de la clave debe ser comparable**), y `V` es un parámetro de tipo con la restricción `int64 | float64` (el tipo del valor puede ser `int64` o `float64`).

Pregunta 2.2: Tipos Genéricos a) ¿Pueden los "structs" ser genéricos en Go? Si es así, proporcione un ejemplo conceptual.

a) Sí, los "structs" pueden ser genéricos en Go. Las fuentes muestran un ejemplo de una lista genérica `List[T any]` y un árbol binario `Tree[T any].` El tipo `T` dentro de los corchetes indica que el `struct` puede operar con cualquier tipo de dato que se le especifique al momento de su instanciación. Ejemplo conceptual basado en las fuentes:

```
    type List[T any] struct { // List es un tipo genérico que puede
contener cualquier tipo T
        first, last *node[T]
    }
    type node[T any] struct { // node también es genérico para contener T
        val T
        next *node[T]
    }
```

b) ¿Qué "type constraints" son comúnmente utilizados en Go y qué representan?

b) "type constraints" comunes: * `any`: Representa cualquier tipo de dato. * `comparable`: Representa tipos que tienen definida la comparación por igualdad (`==`). * `Stringer`: Representa cualquier tipo que implemente la interfaz `Stringer` (es decir, que tenga el método `String() string`). * **Uniones de tipos específicos**, como `int | int16 | int32 | int64 | int8 | float32 | float64`, que restringen el parámetro de tipo a uno de esos tipos numéricos.

Sección 3: Punteros

Pregunta 3.1: Concepto de Puntero a) ¿Qué es un puntero en Go? ¿Cuál es su "zero value"?

a) Un **puntero** es la dirección de memoria de un contenido de cierto tipo. Su "zero value" es `nil`.

b) ¿Cuáles son los operadores asociados con los punteros y qué hace cada uno?

b) Los operadores asociados son: "`&`" (operador de dirección): Devuelve la dirección de memoria de una variable. Ejemplo**:** `p = &i` (puntero `p` apunta a la dirección de `i`). "`*`" (operador de desreferencia): Accede al valor almacenado en la dirección de memoria a la que apunta el puntero. Ejemplo: `fmt.Println(*p)` (imprime el valor al que apunta `p`).

c) ¿Cómo se utiliza la función `new(T)` para trabajar con punteros?

c) La función `new(T)` asigna memoria para una variable de tipo `T` y devuelve un puntero a esa memoria, inicializando el valor con su "zero value". Por ejemplo, `p := new(int)` crea un puntero a un entero inicializado en `0`.

Pregunta 3.2: Punteros en Funciones y Métodos a) Go utiliza el "pasaje por valor" para los parámetros de funciones. ¿Cómo se aplica esto a los punteros cuando se pasan como parámetros?

a) Los parámetros a funciones en Go son "por valor". Esto significa que una copia del puntero se pasa a la función. Sin embargo, aunque el puntero en sí es una copia, su contenido (el valor al que apunta) puede ser modificado a través de la desreferencia (`*xPtr = 0`).

b) Explique la diferencia entre un receptor de valor (`func (c Celsius) String() string`) y un receptor de puntero (`func (f *MyFloat) Scale(s float64)`) en la definición de métodos. ¿Cuándo se usaría uno u otro?

b) * Un **receptor de valor** (`func (c Celsius) String() string`) actúa como un parámetro por valor. Una copia del valor del receptor se pasa al método. Esto significa que el método **no puede modificar** el valor original del receptor. Se usa cuando el método solo necesita leer o realizar cálculos sin alterar el estado del objeto.
* Un **receptor de puntero** (`func (f *MyFloat) Scale(s float64)`) recibe la dirección de memoria del valor. Esto permite que el método **modifique** el valor original al que apunta el receptor. Se usa cuando el método necesita alterar el estado del objeto (ej. `Scale` que modifica `f`).

c) ¿Puede un receptor (receiver) de un método ser `nil`? Si es así, ¿qué implicaciones tiene?

c) Sí, un receptor puede ser `nil`. Las fuentes muestran un ejemplo donde un método (`Add` en `MySlice`) maneja explícitamente el caso en que su receptor (`*MySlice`) sea `nil` antes de intentar desreferenciarlo. Si el método no maneja esta condición y se intenta desreferenciar un puntero `nil`, causará un "runtime error" (panic).

Sección 4: Pasaje de Parámetros

Pregunta 4.1: Modelo de Pasaje de Parámetros a) Describa el modelo general de pasaje de parámetros en Go.

a) Go utiliza un modelo de **pasaje de parámetros "por valor"**. Esto 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, no sobre las variables originales.

b) Explique las implicaciones de este modelo para el pasaje de Arrays y structs a funciones. ¿Se modifican los valores originales dentro de la función llamada?

b) Para **Arrays**, si se pasa un array a una función, se pasa una copia completa del array. Cualquier modificación realizada sobre este array dentro de la función no afectará al array original. De manera similar, cuando se pasa un **struct** a una función, se pasa una copia del struct. Las modificaciones a los campos del struct dentro de la función llamada no afectarán al struct original fuera de la función.

c) ¿Cómo se puede lograr un comportamiento de "paso por referencia" o modificación de un valor externo a la función, dado el modelo de pasaje de parámetros de Go?

c) Para lograr un comportamiento donde la función pueda modificar un valor externo, se deben pasar **punteros** a esos valores como parámetros. Aunque el puntero en sí es pasado por valor (una copia del puntero), el contenido al que apunta puede ser desreferenciado y modificado dentro de la función, afectando así el valor original. Esto también se aplica a los receptores de puntero en los métodos.

Sección 5: Slices

Pregunta 5.1: Concepto y Creación de Slices a) ¿Qué es un "slice" en Go y en qué se diferencia fundamentalmente de un "array"?

a) Un **slice** es un "segmento" de un array. A diferencia de un array, cuya longitud es parte de la definición de su tipo y es fija, la **longitud de un slice puede cambiar**. Los Slices son indexables y tienen una longitud.

b) Mencione y describa tres formas de crear un slice en Go.

b) Hay tres formas de crear un slice: * **A partir de un array existente**: Se puede crear un slice "cortando" una porción de un array, como `s := a[2:4]`. * **Usando un literal de slice**: Se puede declarar y inicializar directamente un slice, como `s3 := []int{1, 2, 3}`. Un slice vacío también se puede declarar como `var s1 []int` o `s2 := []int{}`. * **Usando la función `make()`**: Se utiliza `make([]Tipo, longitud, capacidad)` o `make([]Tipo, longitud)`. Por ejemplo, `s1 := make([]int, 5, 10)` crea un slice de 5 elementos con una capacidad de 10, y `s2 := make([]int, 5)` crea un slice de 5 elementos con una capacidad igual a su longitud (5).

c) ¿Qué representan `len()` y `cap()` en el contexto de los slices?

c) En el contexto de los slices: * `len()`: Devuelve la **longitud** actual del slice, es decir, el número de elementos que contiene. * `cap()`: Devuelve la **capacidad** del slice, que es el número máximo de elementos que puede contener el slice sin tener que reasignar la memoria subyacente (es decir, la longitud del array subyacente desde el inicio del slice).

Pregunta 5.2: Comportamiento y Operaciones de Slices a) Explique cómo los Slices son "referencias a los Arrays subyacentes" y proporcione un ejemplo de cómo una modificación en un slice puede afectar a otro slice o al array original.

a) Los Slices son **referencias** a los Arrays subyacentes. Esto significa que múltiples Slices pueden apuntar al mismo array subyacente. Si un valor es modificado a través de un slice, esa modificación será visible a través de cualquier otro slice que comparta el mismo array subyacente, así como en el propio array subyacente. * Ejemplo: `a :=int{10, 11, 12, 13, 14, 15}` y `s := a[2:4]`. Si luego se hace `s = 31`, el array original `a` también se verá afectado, y `a` se convertirá en `31`.

b) Describa la función `append()` y cómo afecta la longitud y capacidad de un slice.

b) La función `append(slice []Type, elems ...Type) []Type` se utiliza para agregar elementos a un slice. Retorna un nuevo slice que contiene los elementos originales más los nuevos. Si la capacidad subyacente del slice es insuficiente para acomodar los nuevos elementos, Go asignará un nuevo array subyacente de mayor tamaño y copiará los elementos existentes, lo que puede cambiar la capacidad y, en algunos casos, el array subyacente del slice.

c) ¿Cómo se itera sobre un slice utilizando `for range` y cuáles son las opciones para acceder a índices y valores?

c) Se itera sobre un slice utilizando la estructura `for range`. Las opciones para acceder a índices y valores son: * `for index, elem := range slice`: Accede tanto al índice como al elemento en cada iteración. * `for _, elem := range slice`: Accede solo al elemento, **ignorando el índice (usando `_` para el índice).** * `for i, _ := range slice`: Accede solo al índice, **ignorando el elemento (usando `_` para el elemento)**. * `for i := range slice`: Accede solo al índice.

Sección 6: Maps

Pregunta 6.1: Concepto y Características de Maps a) ¿Qué es un "map" en Go y cuáles son sus características principales?

a) Un **map** es una colección no ordenada de pares **clave-valor**. También se les conoce como arreglos asociativos, tablas hash o diccionarios. Sus características principales son: *** No permite claves duplicadas**. *** Es una colección no ordenada**. *** Su valor por defecto es `nil`. * Se puede agregar, modificar y eliminar elementos, excepto si el map es `nil`.**

b) ¿Qué tipos de datos están permitidos como claves en un map? ¿Cuáles no están permitidos?

b) Los tipos de clave permitidos son aquellos que tienen definida la comparación por igualdad (`==`), osea son comparables. Esto incluye `booleans`, `numbers` (enteros, flotantes, complejos), `strings`, y `arrays`. Los tipos que **no están permitidos** como claves son `slices`, `maps` y `functions`.

c) ¿Cuál es el "zero value" de un map? ¿Se pueden agregar elementos a un map con su "zero value"?

c) El "zero value" de un map es `nil`. No se pueden agregar elementos a un map que sea `nil`; intentar hacerlo resultará en un "ERROR EN TIEMPO DE EJECUCIÓN". Para agregar elementos, el map debe ser inicializado, por ejemplo, con `make` o un literal.

Pregunta 6.2: Operaciones y Comportamiento de Maps a) ¿Cómo se agrega, modifica y elimina un elemento en un map? ¿Cómo se recupera un valor y cómo se puede verificar si la clave existe?

a) * **Agregar o modificar**: `m[key] = value`. Si la clave existe, el valor se modifica; si no, se agrega un nuevo par clave-valor. * **Eliminar**: `delete(m, key)`. * **Recuperar valor**: `elem = m[key]`. Si la clave no está presente, `elem` tomará el "zero value" del tipo correspondiente del valor. * **Verificar si la clave existe**: `elem, ok = m[key]`. Esta sintaxis especial devuelve el valor (`elem`) y un booleano (`ok`). `ok` será `true` si la clave existe y `false` si no.

b) ¿Cómo se itera sobre un map en Go? ¿El orden de iteración está garantizado?

b) Se itera sobre un map utilizando la sentencia `for range`, similar a como se hace con arrays o slices, pero devuelve la clave y el valor en cada iteración. Por ejemplo: `for k, v := range m { ... }`. El **orden de iteración no está garantizado**; los maps son colecciones no ordenadas