



Validación Semántica

Lenguaje GO

Marcial Parra Israel
Pérez Rendón Arturo
Juárez Castillo Andrés

Índice

Introducción	2
Errores Semánticos	3
¿Qué son?.....	3
¿Qué es la semántica?.....	3
Error # 1	4
Error # 2.....	5
Error # 3.....	6
Error # 4.....	7
Error # 5.....	8
Error # 6.....	9
Error # 7.....	10
Error # 8.....	11
Error # 9.....	12
Funcionamiento del tipo Booleano # 10	13
Bibliografía	14

Introducción

¿Qué es GO?

Go es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C. Ha sido desarrollado por Google.

Actualmente está disponible en formato binario para los sistemas operativos Windows, GNU/Linux, FreeBSD y Mac OS X, pudiendo también ser instalado en estos y en otros sistemas con el código fuente.

Go es un lenguaje de programación compilado, concurrente, imperativo, estructurado, orientado a objetos —de una manera bastante especial— y con recolector de basura que de momento está soportado en diferentes tipos de sistemas UNIX, incluidos Linux, FreeBSD y Mac OS X.

Errores Semánticos

¿Qué son?

Los errores pueden ser sintácticos, semánticos o lógicos. ... Un error semántico se produce cuando la sintaxis del código es correcta, pero la semántica o significado no es el que se pretendía.

¿Qué es la semántica?

Son el conjunto de reglas que proporcionan el significado de una sentencia o instrucción de cualquier lenguaje de programación.

Go es un lenguaje realmente nuevo, por lo cual no posee todavía un IDE, pero para nuestra suerte los editores de texto están a la orden del día.

Error # 1

Como en cualquier otro lenguaje la duplicidad de las variables siempre será un error semántico, a pesar de que Go si diferencia las letras mayúsculas de las minúsculas, escribirlas exactamente igual (como se muestra en la sig. imagen):

```
func main() {  
    fmt.Println("\n\n\t#####Error1#####")  
    var a int = 5  
    var a int = 5  
    //var a float64 = 2.00  
    var b int = 6  
    var A float64 = 2.00  
    var c string = "Karla Diaz<3"  
    fmt.Println(a,b,c)  
    fmt.Println(A)  
}
```

Nos dirá un error como el siguiente, donde nos indica que la variable ya había sido declarada anterior mente

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> go run ErroresSemanticos.go  
# command-line-arguments  
.\ErroresSemanticos.go:11: a redeclared in this block  
    previous declaration at .\ErroresSemanticos.go:10  
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> |
```

Error # 2

Para este segundo error ocupare la declaración de las variables. La forma correcta de declarar una variable es la siguiente: **var AX7 string** (escribir la palabra “var” seguido del nombre de la variable y por último el tipo de la variable, pero aunque Go es un lenguaje de tipado estático nos permite el duck typing, este nos permite declarar variables sin el tipo ya que el compilador se encarga que dependiendo el valor que ingresemos decidirá el tipo de variable que es. Ejemplo:

```
/*#####Error2#####  
Intentar cambiar el tipo de variable*/  
fmt.Println("\n\n\n\n\n\n\n\n\n\n\t#####Error2#####")  
var AX7 string  
novia := "Karla Diaz 2"  
fmt.Println(novia)  
novia = "Not found"  
fmt.Println(novia)  
novia = 24
```

Como se observa en la imagen esta la forma clásica de declarar la variable y la nueva forma, pero por obvias razones para cualquier persona podría decir “ como esta variable no tiene un tipo en específico podría estar cambiando el tipo de variable que es “ pero no es así, en el primer valor que asignes a esta variable, el compilador asignara que tipo es, por lo cual si después decides intentara cambiar este valor como por ejemplo en la imagen vemos que la variable **novia** inicia teniendo una cadena de caracteres para después asignarle un valor entero, nos arrojará un error

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> go run ErroresSemanticos.go  
# command-line-arguments  
.\ErroresSemanticos.go:41: cannot use 24 (type int) as type string in assignment  
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> |
```

Diciéndonos que no podemos usar valor entero ya que anteriormente esa variable había sido asignada como tipo string

Error # 3

En cualquier otro lenguaje de programación el no utilizar una variable únicamente nos genera un pequeño warning (Advertencia) indicándonos que esta no está siendo utilizada, pero en Go no es este caso, si se declaran variables (Ejemplo):

```
/*#####Error3#####  
EL simple hecho de no Utilizar una variable */  
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error3#####")  
Rivera := "CRACK"  
h := "ss"  
fff := 2  
hh := "s"
```

Y estas no son utilizadas de ninguna manera, el compilador nos mandara el siguiente error:

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\pr  
> go run ErroresSemanticos.go  
# command-line-arguments  
.\ErroresSemanticos.go:50: Rivera declared and not used  
.\ErroresSemanticos.go:51: h declared and not used  
.\ErroresSemanticos.go:52: fff declared and not used  
.\ErroresSemanticos.go:53: hh declared and not used  
  
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\pr  
> |
```

Que las variables han sido declaradas, pero no están siendo utilizadas. Esto es muy bueno ya que nos permite no despreciar espacios en memoria.

Error # 4

Las operaciones aritméticas, en este caso Go realmente es especial ya que a comparación de cualquier otro lenguaje, solo nos permite las operaciones si las variables son del mismo tipo, por consiguiente si nosotros intentamos como sumar 2 tipos diferentes. Ejemplo:

```
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error4#####")
/*#####Error4#####
operaciones de variables*/
var pro1 int = 24
//var pro2 string = "hola"
//var pro2 int = 147
var pro3 float64 = 26
suma := 0
suma = pro1 + pro3
fmt.Println("La sum es:" , suma)
```

Que intentemos sumar un tipo entero con uno flotante, nos dirá el siguiente error:

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> go run ErroresSemanticos.go
# command-line-arguments
.\ErroresSemanticos.go:68: invalid operation: pro1 + pro3 (mismatched types int and float64)

Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> |
```

Que los valores enteros y flotantes no son compatibles

Error # 5

Go nos permite intercambiar los valores entre diferentes variables pero como en el anterior error, únicamente es posible esto si los tipos son los mismos.

```
/*#####Error5#####  
Intercambiar datos contenidos en variables*/  
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error5#####")  
var xd1 int = 300  
// var xd2 int = 600  
// xd1, xd2 = xd2, xd1  
// fmt.Println(xd1 , xd2)  
var xd3 string = "X"  
xd1, xd3 = xd3, xd1  
fmt.Println(xd1 , xd3)
```

Si intentamos intercambiar valores enteros y de cadenas de caracteres entre las variables

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> go run ErroresSemanticos.go  
# command-line-arguments  
.\ErroresSemanticos.go:81: cannot use xd3 (type string) as type int in assignment  
.\ErroresSemanticos.go:81: cannot use xd1 (type int) as type string in assignment  
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> □
```

Nos dirá que no es posible darle ese valor por el tipo que ya se les había asignado

Error # 6

Utilizar variables inexistentes o no declaradas:

```
/*#####Error6#####  
Variables inexistentes o no declaradas en la misma func */  
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error6#####")  
var nombre2 string="Rodrigo"  
fmt.Println("El nombre es ",nombre3)  
  
} //main  
/*#####Error6#####  
Variables inexistentes o d */  
func imprimir() {  
    fmt.Println("El nombre es " nombre2)  
    |  
  
}
```

En el 1er caso (donde esta subrayado de rojo) se está mandando a llamar una variable no declarada, y en el 2do (amarillo) se está mandando a llamar una variable pero no se encuentra dentro de el mismo método


```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> go run ErroresSemanticos.go  
# command-line-arguments  
.\ErroresSemanticos.go:149: undefined: nombre3  
.\ErroresSemanticos.go:156: undefined: nombre2  
  
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1  
> |
```

En ambos casos las marca error como indefinidas una por no existir y la otra que se encuentra en un método distinto

Error # 7

Los arreglos cuando se declaras es algo fundamental definir de cuantas posiciones será este, como se muestra en la siguiente imagen:

```
/*#####Error7#####*/
Arreglos, intentar acceder a un indice no creado */
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error7#####")
var arreglo [3]int
fmt.Println(arreglo)
arreglo[0] = 96
fmt.Println(arreglo[4])
```



En este caso se ha declarado un arreglo de 3 posiciones, pero que sucederá si intentamos acceder a la 4ta posición.

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> go run ErroresSemanticos.go
# command-line-arguments
.\ErroresSemanticos.go:103: invalid array index 4 (out of bounds for 3-element array)

Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> |
```

Nos dirá un error donde marca que este arreglo solo está hecho para 3 posiciones.

Error # 8

El funcionamiento de la sentencia if es similar a la de cualquier otro lenguaje de programación, sin embargo como ya es costumbre únicamente puede comparar variables del mismo tipo.

```
fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error8#####")
var rd int = 4
//var rd2 int = 8
var rd3 bool = true
if rd2 > rd3 {

    fmt.Println("8 es mayor que 4")
}
```

Por consiguiente si intentamos en un ciclo if comparar 2 tipos diferentes.

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> go run ErroresSemanticos.go
# command-line-arguments
.\ErroresSemanticos.go:124: invalid operation: rd > rd3 (mismatched types int and bool)

Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> |
```

Nos dirá el mismo error que las variables no son compatibles

Error # 9

Si de algo carece el lenguaje Go es de las estructuras de control, ya que solo posee las sentencias IF y FOR.

En Go hay 3 formas de hacer un for pero ocuparemos la forma clásica, que contiene la inicialización de una variable, una condición y su incremento

```
const s string = "constant"
func main(){
    //var contador float64 = 0
    fmt.Println("\n\n\n\n\n\n\n\n\n\t#####Error9#####")
    var k bool = true
    //var j int = 5
    for i:=1 ; i < k ; i++ {
        /* for j:=1; j < 5 ; i++){
            fmt.Println(j)
            i= i-1
            contador = contador +1
            fmt.Println("itera no:", contador)

        }*/
        fmt.Println(i)
```

Como de costumbre se va a comparar en la condición valores de diferentes tipos.

```
Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> go run 2.go
# command-line-arguments
.\2.go:11: invalid operation: i < k (mismatched types int and bool)

Israel@DESKTOP-EONGGT8 C:\Users\Israel\Documents\Golang\programa1
> |
```

Dándonos como resultado el mismo error de los tipos no compatibles.

Funcionamiento del tipo Booleano # 10

A diferencia de otros lenguajes, Go no consideran como 0 y 1 los valores para true y false.

A continuación un pequeño código sobre el tipo bool

```
}//if
/*#####Error10#####
Funcionamiento del booleano*/

fmt.Println("\n\n\n\n\n\n\n\n\n\t#####10#####")
var resul bool
resul = 3 > 5 🟡
fmt.Println(resul)
resul = (5 < 6)&&(3 > 1) 🟡
fmt.Println("\n", resul)
```

Primeramente asignamos una comparación sencilla a la variable booleana, y después asignamos una operación lógica, en este caso **and**

```
#####10#####
1ra false
2da true
```

El primer resultado es false ya que la comparación nos dice que 3 es mayor que 5 lo cual es falso.

En la 2da el resultado es true ya que la tabla de verdad del operador lógico **and** nos dice que sí y solo si ambas condiciones se cumplen esta será verdadera, en este caso 5 es menor que 6 y 3 mayor que 1

Bibliografía

www.goconejemplos.com

<https://github.com/byscuellar/golang-go-ejemplos>

https://github.com/Gyga8K/Curso-Go-de-0-a-100/blob/master/16_Ejercicio_1/main.go

Enlace al video:

<https://www.youtube.com/watch?v=St2kgJr5ZvI&feature=youtu.be>