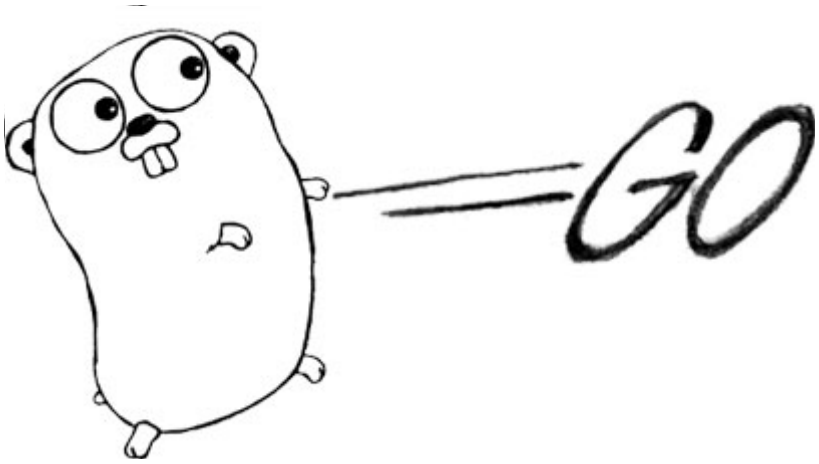
FACULTAD DE INGENIERIA
Universidad de Buenos Aires



Lenguaje Go

Informe de la exposición realizada en clase.

Adrián Mouly (92501) - Sebastián Torres (87727)

Tabla de Contenidos

1. Introducción

Go es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C. Ha sido desarrollado por Google y sus diseñadores iniciales son Robert Griesemer, Rob Pike y Ken Thompson. Nace en 2007 pero no toma importancia hasta 2009, cuando es impulsado por una fuerte comunidad de desarrolladores.

El lenguaje fue desarrollado para suplir con los requerimientos de la informática moderna, la cual cambió mucho en la última década.

En el apartado del hardware, el contexto actual es el siguiente: gran importancia del networking; computación en grandes clusters; servicios orientados a cliente/servidor; masificación de procesadores multi-core.

En el apartado del software, los avances de la última década presentaron algunas dificultades a la hora de desarrollar productos. Se pueden mencionar: compilar un programa lleva mucho tiempo; las herramientas de producción son cada vez más lentas y complejas; la cantidad de librerías y dependencias aumenta constantemente; la complejidad de los programas aumentan a medida que se añaden características o se adapta a los cambios de la realidad.

Pese a este avance de los sistemas informáticos, no existía un lenguaje diseñado con estos aspectos en mente. **Go** pretende ocupar ese lugar.

Para cumplir con esta premisa, los objetivos del lenguaje son: aumentar la productividad del programador; compilar rápidamente los programas; ser eficiente en todo momento, desde el diseño de un programa hasta la ejecución del mismo; ofrecer buen soporte de concurrencia y comunicaciones, indispensable para la informática moderna; ofrecer un entorno de ejecución seguro y controlado.

2. Sintaxis básica

2.1. Paquetes

El lenguaje impone ciertos aspectos sobre el diseño de un programa, los cuales deben ser respetados.

Uno de ellos es la utilización de paquetes. Todo programa desarrollado en **Go**, esta formado por paquetes. Un paquete esta compuesto por todos aquellos archivos o subprogramas que componen un programa más grande. Esta característica permite una mejor organización de los archivos, además de una mejor utilización de librerías.

Una aplicación comienza a ejecutarse desde el paquete "main". Por convención, el paquete lleva el nombre de la función principal.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     fmt.Println("Hola Mundo")
7  }
```

En este ejemplo, se muestra la definición de un paquete “main”, el cual importa el paquete “fmt”. Dicho paquete provee las funciones de entrada y salida estándar al lenguaje. En este caso se hace uso de la función “Println”, alojada en el paquete “fmt”, para imprimir por pantalla la frase “Hola Mundo”.

2.2. Funciones

Las funciones en **Go** comienzan con la palabra reservada *func*. Después, nos encontramos el nombre de la función, y una lista de parámetros entre paréntesis. El valor devuelto por una función, si es que devuelve algún valor, se encuentra después de los parámetros. Finalmente, utiliza la instrucción return para devolver un valor del tipo indicado.

Si la función devuelve varios valores, los tipos de los valores devueltos van en una lista separada por comas entre paréntesis.

Los parámetros devueltos por una función son, en realidad, variables que pueden ser usadas si se les otorga un nombre en la declaración. Además, están inicializadas, por defecto, a ”cero” (0, 0.0, false... dependiendo del tipo de la variable).

```
1  func MySqrt(f float) (float, bool) {
2     if f >= 0 {
3        return math.Sqrt(f), true
4     }
5     return 0, false
6  }
7
8  func MySqrt2(f float) (v float, ok bool) {
9     if f >= 0 {
10       v, ok = math.Sqrt(f), true
11    }
12    return          // Retorna v,ok con sus valores
13 }
```

En este ejemplo se crean dos funciones, donde la primer función retorna los resultados de forma explícita, en cambio la segunda función considera los valores de retorno como nuevas variables, retornando estas implícitamente.

Funciones anónimas: Están definidas en tiempo de ejecución. Se ejecutan en el contexto en el que son declaradas. A menudo, son referenciadas por variable para poder ser ejecutadas en otro contexto.

Closures: Se pueden definir como funciones anónimas declaradas en el ámbito de otra función y que pueden hacer referencia a las variables de su función contenedora incluso después de que ésta se haya terminado de ejecutar.

En tiempo de ejecución, cuando la función externa se ejecuta, se forma una clausura, consistiendo en el código de la función interna y referencias a todas las variables de la función externa que son requeridas por la clausura.

Una clausura puede ser usada para asociar una función con un conjunto de variables "privadas", que persisten en las invocaciones de la función. El ámbito de la variable abarca sólo al de la función definida en la clausura, por lo que no puede ser accedida por otro código del programa. No obstante, la variable mantiene su valor

de forma indefinida, por lo que un valor establecido en una invocación permanece disponible para la siguiente.

Para ejemplificar ambos conceptos juntos:

```
1  func sumador(){
2     var x int
3     return   func(delta int) int {
4                    x += delta
5                    return x
6              }
7  }
8
9  var f = sumador() // f es una función ahora.
10 fmt.Printf(f(1))
11 fmt.Printf(f(20))
12 fmt.Printf(f(300))
```

Este ejemplo imprime los valores 1, 21 y 321, ya que irá acumulando los valores en la variable x de la función f.

Puede observarse que la clausura se forma cuando se ejecuta la función sumador().

A la variable “f” se le asigna la función anónima que devuelve la función sumador(), con lo cual se transforma automáticamente en una función con las características de la función anónima y las variables locales de la función sumador(), es decir una función con estado interno similar a un objeto.

2.3. Estructuras de control

2.3.1. If/Else

La sentencia condicional “if” permite ejecutar una rama de código u otra dependiendo de la evaluación de la condición.

Esta sentencia, es similar a la existente en otros lenguajes, pero tiene el problema de que el “else” debe estar en la misma línea que la llave de cierre del “if”.

Además admite instrucciones de inicialización, separadas por un punto y coma. Esto es muy útil con funciones que devuelvan múltiples valores.

Si una sentencia condicional “if” no tiene ninguna condición, significa “true”, lo que en este contexto no es muy útil pero sí que lo es para “for” o “switch”.

Algunos ejemplos de la sintaxis del if/else:

```
1  if x < 5 { f() }
2
3  if x < 5 { f() } else if x == 5 { g() }
4
5  if v := f(); v < 10 {
6     fmt.Printf("\%d es menor que 10\n", v)
7  } else {
8     fmt.Printf("\%d no es menor que 10", v)
9  }
```

2.3.2. For

En Go sólo tenemos un tipo de bucle, a diferencia de otros lenguajes que suelen poseer varios. Todo lo que implique un bucle, en **Go** se ejecuta con una instrucción “for”. Una de las cosas más llamativas de los bucles en **Go**, es la posibilidad de usar asignaciones a varias variables, y hacer un bucle doble en una sola instrucción.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6     sum := 0
7
8     for i := 0; i < 10; i++ {
9        sum += i
10    }
11
12    fmt.Println(sum)
13 }
```

2.3.3. Switch

El “switch” tiene la misma apariencia que en la mayoría de los lenguajes y, aunque es en apariencia similar al de C, tiene varias diferencias:

- Las expresiones no tienen porque ser constantes ni siquiera números.
- Las condiciones no se evalúan en cascada automáticamente.
- Para conseguir evaluación en cascada, se puede usar la palabra reservada “fallthrough”.
- Múltiples casos que ejecuten una misma secuencia de instrucciones, pueden ser separados por comas.

```
1  switch count {
2     case 4, 5, 6: error()
3     case 3: a *= v; fallthrough
4     case 2: a *= v; fallthrough
5     case 1: a *= v; fallthrough
6     case 0: return a*v
7     default: fmt.Printf("No es ninguno")
8  }
```

El “switch” de **Go** es más potente que el de C. Además de que las expresiones pueden ser de cualquier tipo, sabemos que la ausencia de la misma significa “true”. Así, podemos conseguir una cadena “if −else” con un “switch”.

Veamos un ejemplo:

```
1  // If-else
2  a, b := 3, 5;
3
4  // Un switch vacío significa true
5  switch {
6     case a < b: return -1
7     case a == b: return 0
8     case a > b: return 1
9  }
10
11 // Este último switch sería equivalente a
12 switch a, b := 3, 5; { ... }
```

2.3. Tipos

2.3.1 Tipos Numéricos

Existen tres tipos numéricos: Enteros, enteros sin signo y números flotantes.

Cada uno de estos tipos tiene asociadas una serie de variantes dependiendo del número de bits en el que sea almacenado. Veamos un cuadro resumen con los tipos numéricos existentes:

Enteros	Enteros sin signo	Flotantes
int	uint	float
int8	uint8=byte	
int16	uint16	
int32	uint32	float32
int64	uint64	float64

También existe el tipo "uintptr", que sirve para almacenar números enteros lo suficientemente grandes como para necesitar un puntero.

Como puede deducirse de la tabla anterior, el número que acompaña a cada nombre de tipo, es el número de bits que ocupa en memoria. Así podemos observar que los números flotantes no tienen representaciones válidas con 8 y 16 bits.

Los tipos que no tienen asociado ningún número, int, uint y float, se representan con un número de bits igual al ancho de la palabra de la máquina en la que se ha compilado el código.

De esta forma se puede llegar a pensar que en un ordenador de 32-bit, los tipos int y int32 son equivalentes, pero no es así. Todos los tipos de la tabla son distintos.

Debido a que **Go** es un lenguaje con tipado fuerte, no existe conversión implícita de tipos.

2.3.2. Tipos Booleanos

El tipo bool define el tipo booleano usual, con dos constantes predefinidas que son: “true” y “false”.

Hay que tener en cuenta, que a diferencia de otros lenguajes, en Go los punteros y los enteros no son booleanos.

2.3.3. Tipo String

El tipo string representa arrays invariables de bytes, o lo que es lo mismo, texto. Los strings están delimitados por su longitud, no por un carácter nulo como suele ocurrir en la mayoría de los lenguajes. Esto hace que el tipo string sea mucho más seguro y eficiente.

Toda cadena de caracteres representada por el lenguaje, incluidas las cadenas de caracteres literales, tienen como tipo string.

Como se ha dicho en el primer párrafo, y al igual que ocurre con los números enteros, los strings son invariables. Esto significa, que se pueden reasignar variables de tipo string para que contengan otros valores, pero los valores de una variable de este tipo no se pueden modificar.

De la misma forma que 5 siempre es 5, ”Hola” siempre es ”Hola”.

2.3.4. Tipo Struct

Las structs son un tipo de datos que contienen una serie de atributos y permiten crear tipos de datos más complejos. Su sintaxis es muy común, y se pueden declarar de dos formas:

```
1  // Forma resumida
2  var p struct { x, y float }
3
4  // Forma más usual
5  type Punto struct { x, y float }
6  var p Punto
```

Como casi todos los tipos en **Go**, las structs son valores, y por lo tanto, para lograr crear una referencia o puntero a un valor de tipo struct, usamos el operador new(StructType).

En el caso de punteros a estructuras en Go, no existe la notación − >, sino que Go ya provee la indirección al programador.

```
1  type Point struct { x, y float }
2
3  var p Point
```

```
4  p.x = 7
5  p.y = 23.4
6
7  var pp *Point = new(Point)
8  *pp = p
9  pp.x = Pi // equivalente a (*pp).x
```

Ya que los structs son valores, se puede construir un struct a “cero” simplemente declarando. También se puede realizar reservando su memoria con new().

```
1  var p Point // Valor a "cero"
2  pp := new(Point); // Reserva de memoria idiomática
```

Al igual que todos los tipos en Go, los structs también tienen sus literales correspondientes:

```
1  p = Point { 7.2, 8.4 }
2  p = Point { y:8.4, x:7.2 }
3  pp := &Point { 23.4, -1 } // Forma correcta idiomáticamente
```

2.3.5. Tipo Array

Los arrays son una estructura de datos que permiten tener una serie de datos del mismo tipo distribuidos uniformemente en un bloque de memoria.

Los arrays de Go son más cercanos a los de Pascal que a los de C. Más adelante, veremos los slices que son más parecidos a los arrays de C.

La declaración de los arrays se realiza con la palabra reservada “var” acompañada del nombre de la variable, y el tamaño y tipo de datos que tendrá el array. Todos los arrays deben tener un tamaño explícito.

```
1  var ar [3]int
```

La anterior declaración declara un array de nombre “ar” con capacidad para 3 n´meros enteros. Ya que no han sido inicializados, por defecto serán 0.

Si en algún momento queremos averiguar el tamaño de un array, podemos hacer uso de la función len().

Los arrays son valores, no punteros implícitos como ocurre en C. De todas formas, se puede obtener la dirección de memoria donde se ha almacenado un array, que podría servir para pasar un array de forma eficiente a una función. Veamos un ejemplo:

```
1  func f(a [3]int) {
```

```
2     fmt.Println(a)
3  }
4
5  func fp(a *[3]int) {
6     fmt.Println(a)
7  }
8
9  func main() {
10    var ar [3] int
11    f(ar) // Pasa una copia de ar
12    fp(&ar) // Pasa un puntero a ar
13 }
```

La salida del ejemplo anterior sería la siguiente:

```
1 [0 0 0]
2 &[0 0 0]
```

La función Println conoce la estructura de un array y cuando detecta uno, lo imprime de forma óptima.

Los arrays también tienen su propio literal, es decir, su forma de representar el valor real de un array. Veamos un ejemplo:

```
1  // Array de 3 enteros
2  [3]int { 1, 2, 3 }
3
4  // Array de 10 enteros, los 3 primeros no nulos
5  [10]int { 1, 2, 3 }
6
7  // Si no queremos contar el número de elementos
8  // '...' lo hace por nosotros.
9  [...]int { 1, 2, 3 }
10
11 // Si no se quiere inicializar todos,
12 // se puede usar el patrón 'clave:valor'
13 [10]int { 2:1, 3:1, 5:1, 7:1 }
```

2.3.6. Tipo Slice

Un slice es una referencia a una sección de un array. Los slices se usan más comúnmente que los arrays.

Un slice se declara como un array nada más que este no tiene tamaño asociado:

```
1 var a []int
```

Si queremos obtener el número de elementos que posee un slice, recurrimos a la función len(), de igual forma que si fuera un array.

Un slice puede crearse “troceando” un array u otro slice.

```
1  a = ar[7:9]
```

Esta instrucción nos generará un slice a partir de un array, tomando los índices 7 y 8 del array. El número de elementos devuelto por len(a) == 2, y los índices válidos para acceder al slice “a”, son 0 y 1.

De igual forma, podemos inicializar un slice asignándole un puntero a un array:

```
1  a = &ar // Igual que a = ar[0:len(ar)]
```

Según un reciente cambio en la sintaxis referente a los slices, no es necesario poner implícitamente los dos valores, de comienzo y fin, sino que poniendo únicamente el valor de comienzo, nos creará un slice hasta el final del array o slice que estamos referenciando.

```
1  a = ar[0:] // Igual que a = ar[0:len(ar)]
2  b = ar[5:] // Igual que b = ar[5:len(ar)]
```

Al igual que los arrays, los slices tienen sus literales correspondientes, que son iguales pero no tienen tamaño asociado.

```
1  var slice = []int {1, 2, 3, 4, 5}
```

Lo que esta instrucción hace, es crear un array de longitud 5, y posteriormente crea un slice que referencia el array.

Podemos también reservar memoria para un slice (y su array correspondiente) con la función predefinida make():

```
1  var s100 = make([]int, 100) // slice: 100 enteros
```

¿Por qué usamos make() y no new()? La razón es que necesitamos construir un slice, no sólo reservar la memoria necesaria. Hay que tener en cuenta que make([] int) devuelve []int, mientras que new([]int) devuelve *[]int).

Un slice se refiere a un array asociado, con lo que puede haber elementos más allá del final de un slice que están presentes en el array. La función cap() (capacity) nos indica el número de elementos que el slice puede crecer. Veamos un ejemplo:

```
1  var ar = [10]int {0,1,2,3,4,5,6,7,8,9}
2  var a = &ar[5:7] // Referencia al subarray {5, 6}
3
4  // len(a) == 2 y cap(a) == 5. Se puede aumentar el slice:
5  a = a[0:4] // Referencia al subarray {5,6,7,8}
6
7  // Ahora: len(a) == 4. cap(a) == 5
```

¿Cómo es posible que cap(a) == 5? El aumento del slice puede ser de hasta 5 elementos. Teniendo en cuenta que si aumentamos el slice con a[0:5], conseguiríamos un subarray con los valores del 5 al 9.

Los slices pueden ser utilizados como arrays crecientes. Esto se consigue reservando memoria para un slice con la función make() pasándole dos números - longitud y capacidad - y aumentándolo a medida que crezca:

```
1  var sl = make([]int, 0, 100) // Len == 0, cap == 100
2
3  func appendToSlice(i int, sl []int) []int {
4     if len(sl) == cap(sl) { error(...) }
5     n := len(sl)
6     sl = sl[0:n+1] // Aumentamos el tamaño una unidad
7     sl[n] = i
8     return sl
9  }
```

La longitud de “sl” siempre será el número de elementos y crecerá según se necesite. Este estilo es mucho más “barato” e idiomático en **Go**.

Para terminar con los slices, hablaremos sobre lo “baratos” que son. Los slices son una estructura muy ligera que permiten al programador generarlos y aumentarlos o reducirlos según su necesidad. Además, son fáciles de pasar de unas funciones a otras, dado que no necesitan una reserva de memoria extra.

Hay que recordar que un slice ya es una referencia de por sí, y que por lo tanto, el almacenamiento en memoria asociado puede ser modificado.

2.3.7. Tipo Map

Los maps son otro tipo de referencias a otros tipos. Los maps nos permiten tener un conjunto de elementos ordenados por el par “clave:valor”, de tal forma que podamos acceder a un valor concreto dada su clave, o hacer una asociación rápida entre dos tipos de datos distintos.

Veamos cómo se declararía un map con una clave de tipo string y valores de tipo float:

```
1  var m map[string] float
```

Este tipo de datos es análogo al tipo *map<string,float> de C++ (Nótese el *). En un map, la función len() devuelve el número de claves que posee.

De la misma forma que los slices una variable de tipo map no se refiere a nada. Por lo tanto, hay que poner algo en su interior para que pueda ser usado. Tenemos tres métodos posibles:

- Literal: Lista de pares “clave:valor” separados por comas.

```
1  m = map[string] float { "1":1, "pi:3.1415 }
```

- Creación con la función make()

```
1  m = make(map[string] float)
```

- Asignación de otro map

```
1  var m1 map[string] float
2  m1 = m // m1 y m ahora referencian el mismo map.
```

2.3.7.1. Indexando un Map

Si tenemos el siguiente Map declarado:

```
1  m = map[string] float { "1":1, "pi":3.1415 }
```

Podemos acceder a un elemento concreto de un map con la siguiente instrucción, que en caso de estar el valor en el map, nos lo devolverá, y sino provocará un error.

```
1  uno:= m["1"]
2  error := m["no presente"] //error
```

Podemos de la misma forma, poner un valor a un elemento. En caso de que nos equivoquemos y pongamos un valor a un elemento que ya ten´a un valor previo, el valor de dicha clave se actualiza.

```
1  m["2"] = 2
2  m["2"] = 3 // m[2] vale 3
```

2.3.7.2. Borrando una entrada

Borrar una entrada de un map se puede realizar mediante una asignación multivalor al map, de igual forma que la comprobación de la existencia de una clave.

```
1  m = map[string] float { "1":1, "pi":3.1415 }
2
3  var value float
4  var present bool
5  var x string = f()
6
7  m [x] = value, present
```

Si la variable “present” es true, asigna el valor v al map. Si “present” es false, elimina la entrada para la clave x. Entonces, para borrar una entrada:

```
1  m[x] = 0, false
```

2.3.8. Conversiones

Como ya se ha mencionado anteriormente, las conversiones entre distintos tipos deben realizarse de manera explícita, por lo que cualquier intento de conversión implícita fallará en tiempo de compilación.

Entonces, convertir valores de un tipo a otro es una conversión explícita, que se realiza como si fuera la llamada a una función.

Veamos algún ejemplo:

```
1  uint8(int_var)    // Truncar al tamño de 8 bits
2  int(float_var)    // Truncar a la fracción
3  float64(int_var)  // Conversión a flotante
```

También es posible realizar conversiones a string, de manera que podemos realizar las siguientes operaciones, entre otras:

```
1  string(0x1234)          // == "\u1234"
2  string(array_de_bytes)  // bytes -> bytes
3  string(array_de_ints)   // ints -> Unicode/UTF-8
```

2.5. Objetos

En este apartado, surge una discusión. Existen distintos consensos sobre la teoría de "orientación a objetos". Si consideramos a los objetos como ententes individuales, con la capacidad de intercambiar información entre ellos, podríamos decir que **Go** presenta estas características a través de los canales y corutinas. En cambio, si consideramos la teoría tradicional de objetos, donde existen mecanismos de clases, herencia, interfaces, entonces **Go** no es orientado a objetos.

Podríamos decir que Go permite programar basándose en objetos, pero de una forma no convencional, o no como la realizan otros lenguajes.

2.5.1. Métodos

Go no posee clases, pero su ausencia no impide que se puedan crear métodos específicos para un tipo de datos concreto. Es posible crear métodos para casi cualquier tipo de datos, siempre y cuando dicho tipo esté en el mismo paquete que sus métodos.

2.5.1.1. Métodos para structs

Los métodos en **Go** se declaran de forma independiente de la declaración del tipo. Estos se declaran como funciones con un receptor explícito. Siguiendo con el ejemplo del tipo Punto:

```
1  type Punto struct { x, y float }
2
3  // Un metodo sobre *Punto
4  func (p *Punto) Abs() float {
5     return math.Sqrt(p.x*p.x + p.y*p.y)
6  }
```

Cabe notar que el receptor es una variable explícita del tipo deseado (no existe puntero a this, sino que hay una referencia explícita al tipo *Punto.

Un método no requiere un puntero a un tipo como receptor, sino que podemos usar un tipo pasado como valor. Esto es más costoso, ya que siempre que se invoque al método el objeto del tipo será pasado por valor, pero aun así es igualmente válido en **Go**.

```
1  type Punto3 struct { x, y float }
2
3  // Un metodo sobre Punto3
4  func (p Punto3) Abs() float {
5     return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
```

```
6 }
```

2.5.1.2. Invocación de métodos

Los métodos se invocan de la misma manera que otros lenguajes orientados a objetos.

```
1  p := &Point { 3, 4 }
2  fmt.Print(p.Abs()) // Imprimirá 5
3
4  //Ejemplo con un tipo de datos que no sea de tipo struct.
5  type IntVector []int
6
7  func (v IntVector) Sum() (s int) {
8     for i, x := range v {
9        s += x
10    }
11    return
12 }
13
14 fmt.Println(IntVector { 1, 2, 3 }. Sum())
```

2.5.1.3. Punteros y valores en los métodos

Go automáticamente indirecciona o derreferencia los valores cuando se invoca un método.

Por ejemplo, aunque un método concreto tenga como receptor el tipo *Punto, se puede invocar al método con un valor direccionable de tipo Point. Entenderemos esto mejor con un ejemplo:

```
1  p1 := Punto { 3, 4 }
2
3  // Azúcar sintáctico para (&p1).Abs()
4  fmt.Print(p1.Abs())
5
6  p3 := &Punto3 { 3, 4, 5 }
7
8  // Azúcar sintáctico para (*p3).Abs()
9  fmt.Print(p3.Abs())
```

2.5.1.4. Atributos anónimos en los métodos

Cuando un atributo anónimo está dentro de un tipo struct, se tiene una herencia de los métodos de ese atributo anónimo.

Este método ofrece una manera simple de emular algunos de los efectos de las subclases y la herencia utilizado por los lenguajes de programación orientados a objetos más comunes.

Veamos un ejemplo de atributos anónimos:

```
1  type Punto struct { x, y float }
2
3  func (p *Punto) Abs() float { ... }
4
5  type PuntoNombre struct {
6     Point
7     nombre string
8  }
9
10 n := &PuntoNombre { Punto { 3, 4 }, "Pitágoras" }
11 fmt.Println(n.Abs()) // Imprime 5
```

La sobreescritura de los métodos funciona exactamente igual que con los atributos:

```
1  type PuntoNombre struct {
2     Point
3     nombre string
4  }
5
6  func (n *PuntoNombre) Abs() float {
7     return n.Punto.Abs() * 100
8  }
9
10 n := &PuntoNombre { Punto { 3, 4 }, "Pitágoras" }
11 fmt.Println(n.Abs()) // Imprime 500
```

2.1.1.5. Visibilidad de atributos y métodos

Go tiene visibilidad local a nivel de paquete.

La forma de escribir una variable o método determina su visibilidad (público o exportado/ privado o local). Si los nombres empiezan con mayúscula son exportados y si empiezan con minúscula son locales.

Las estructuras definidas en el mismo paquete, tienen acceso a cualquier atributo y método de cualquier otra estructura.

Un tipo de datos local puede exportar sus atributos y sus métodos.

2.5.2. Interfaces

2.5.2.1. Introducción a interfaces

Una interfaz es algo completamente abstracto y que por sí solo no implementa nada.

Las interfaces en **Go** son muy similares a las interfaces de Java, y aunque Java posee un tipo interface, **Go** implementa un nuevo concepto denominado 'valor de una interfaz'.

2.5.2.2. Definición de interfaz

Una interfaz es un conjunto de métodos.

Se puede tomar dicha definición de otra forma ya que los métodos implementados por un tipo concreto de datos como struct, conforman la interfaz de dicho tipo.

Ejemplo:

```
1  type Punto struct { x, y float }
2
3  func (p *Punto) Abs() float { ... }
```

Con ese tipo que ya hemos visto anteriormente, podemos definir que su interfaz consta de un único método:

```
1  Abs() float
```

No confundir el interfaz con la declaración de la función, ya que la interfaz abstrae completamente el receptor del mismo.

```
1  func (p *Punto) Abs() float { ... }
```

Si volvemos atrás, se puede observar que teníamos el tipo Punto embebido en un tipo NombrePunto. Este último tendría la misma interfaz.

2.5.2.3. El tipo interface

Un tipo interface es una especificación de una interfaz, es decir, un conjunto de métodos que son implementados por otros tipos.

Ejemplo:

```
1  type AbsInterface interface {
```

```
2     Abs() float // El receptor es implícito
3  }
```

Esta es la definición de una interfaz implementada por Punto, o en nuestra terminología: Punto implementa AbsInterface.

También, siguiendo la misma lógica: NombrePunto y Punto3 implementan AbsInterface.

Ejemplo:

```
1  type MyFloat float
2
3  func (f MyFloat) Abs() float {
4     if f < 0 { return -f }
5     return f
6  }
```

MyFloat implementa AbsInterface a pesar de que el tipo nativo float no lo hace.

Una interfaz puede ser implementada por un n´umero arbitrario de tipos. AbsInterface es implementada por cualquier tipo que tenga un m´etodo Abs() float, independientemente del resto de métodos que el tipo pueda tener. Asimismo, un tipo puede implementar un número arbitrario de interfaces.

Punto implementa, al menos, estas dos interfaces:

type AbsInterface interface { Abs() float }
type EmptyInterface interface { }

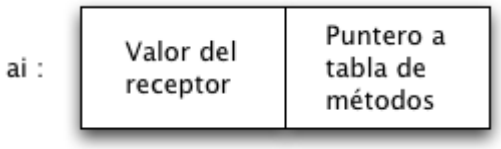
Todos los tipos implementarán una interfaz vacía EmptyInterface.

2.5.2.4. El valor interfaz

Una vez una variable es declarada con un tipo interface, puede almacenar cualquier valor que implemente dicha interfaz.

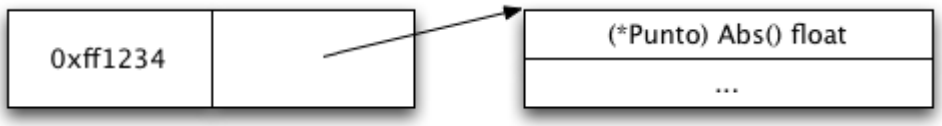
```
1  var ai AbsInterface
2  pp := new(Punto)
3  ai = pp // OK: *Punto tiene Abs()
4  ai = 7. // Error en compilación: float no tiene Abs()
5  ai = MyFloat(-7.) // OK: MyFloat tiene Abs()
6  ai=&Point { 3, 4 }
7  fmt.Printf(ai.Abs()}
```

La variable **ai** no es un puntero, sino que es una estructura de datos de varias palabras:



En distintos momentos de la ejecución del programa posee distintos valores y tipo:

ai = &Punto { 3, 4 } (== (*Punto) (0xff1234)):



2.5.2.5. Polimorfismo

Go utiliza interfaces para aplicar el polimorfismo.

Ejemplo: io.Writer

El paquete io concede una serie de métodos para el programador que pueden tratar la entrada y salida de datos. Para ver de forma más clara cómo son las interfaces, se ha elegido la interfaz io.Wirter por ser bastante representativa y muy usada en Go.

Si echamos un vistazo a la cabecera de la función Fprintf del paquete fmt, observamos que tiene la siguiente forma:

```
1  func Fprintf (w io.Writer, format string, a ...) (n int, error
2  os.Error)
```

Es decir, dicha función lo que recibe es una variable u objeto que sea de tipo io.Writer, no necesariamente un fichero donde escribir.

Como se puede observar, Fprintf devuelve una tupla con el número de caracteres escritos y el error en caso de que hubiera alguno.

La interfaz io.Writer está definida de la siguiente manera:

```
1  type Writer interface {
2     Write (p []byte) (n int, err os.Error)
3  }
```

Lo que quiere decir que todos los objetos que instancien dicha interfaz deberán tener un método Write que reciba un buffer de bytes y devuelva el número de caracteres escritos y un error en caso de que éste exista.

Si vemos el paquete bufio que contiene métodos para realizar una entrada /salida con un buffer de datos, observamos que implementa un nuevo tipo de datos:

```
1  type Writer struct { ... }
```

Implementando dicho tipo de datos (bufio.Writer) el método Writer canónico visto en la interfaz del mismo nombre.

```
1  func (b *Writer) Write (p []byte) (n int, err os.Error)
```

Casi todos los paquetes de entrada / salida así como los que se encargan de la escritura en ficheros utilizan el interfaz Write.

El paquete io tiene declarados 4 interfaces distintos, uno para cada uso distinto:

Reader

Writer

ReadWriter

ReadWriteCloser

2.5.2.6. Contenedores y la interfaz vacía

Los contenedores son estructuras de datos que permiten almacenar elementos de cualquier tipo o de un determinado tipo.

Ejemplo:

```
1  type Element interface {}
2
3  //Vector es el propio contenedor
4  type Vector struct {
5     a []Element
6  }
7
8  // At() devuelve el i-ésimo elemento
9  func (p *Vector) At(i int) Element {
10    return p.a[i]
11 }
```

2.5.2.7. Asertos de tipos

Una vez que se almacena un elemento en un Vector, dicho elemento se almacena como un valor interfaz. Es necesario por lo tanto “desempaquetarlo” para tener el valor original, usando los asertos de tipos (o type assertions). Su sintaxis es:

```
1  valor_interfaz.(tipo_a_extraer)
```

Ejemplo:

```
1  var v vector.Vector
2  v.Set (0, 1234.) // Se guarda como un valor de interfaz
3  i := v.At(0) // Devuelve el valor como interface{}
4  if i != 1234. {} // Error en tiempo de compilación
5  if i.(float) != 1234. {} // OK
6  if i.(int) != 1234 {} // Error en tiempo de ejecución
7  if i.(MyFloat) != 1234. {} // Error: no es MyFloat
```

2.5.2.8. Conversión de una interfaz a otra

Hasta ahora únicamente nos hemos limitado a mover valores normales dentro y fuera de valores interfaz, pero los valores de este tipo que contengan los métodos apropiados, también pueden ser convertidos a otros tipos de interfaces. De hecho, es lo mismo que desempaquetar el valor interfaz para extraer el valor concreto al que se hace referencia, para después volverlo a empaquetar para el nuevo tipo interfaz.

La viabilidad de la conversión depende del valor referenciado, no del tipo original de la interfaz.

Ejemplo de la conversión de intefaces, dadas las siguientes definiciones de variables y tipos:

```
1  var ai AbsInterface
2
3  type SqrInterface interface { Sqr() float }
4
5  var si SqrInterface
6  pp := new(Point) // asumamos que *Point tiene Abs, Sqr
7
8  var empty interface{}
9
10 empty = pp
11
12 // Todo satisface empty
13 ai = empty.(AbsInterface)
14
15 // El valor referenciado implementa Abs(). En cualquier
16 otro caso, produce un fallo en tiempo de ejecución si =
17 ai.(SqrInterface) *Point tiene Sqr() aunque AbsInterface no lo
18 implemente empty = si *Point implementa el conjunto vacío
```

3. Concurrencia

En la informática moderna, la concurrencia y la programación multi-hilos presenta algunas dificultades a la hora de programar o diseñar un programa. Go trata de simplificar el uso de concurrencia, ya que lo considera indispensable para los programas actuales.

Se utiliza el concepto de “goroutine”, abstrayendo la utilización de threads, de los cuales se ocupa el run-time system de forma eficiente.

Una “goroutine” es una función corriendo independientemente en el mismo espacio de memoria de otras “goroutines”.

El run-time system se encarga de separar las rutinas en ejecución en distintos hilos. Cuando una rutina que esta siendo ejecutada se bloquea, el sistema automáticamente mueve la rutina siguiente a otro hilo, de esta forma evita que se paralice la ejecución del programa.

Los hilos no comparten memoria para comunicarse, como hacen otros lenguajes. El proceso es inverso: para compartir información entre procesos, estos deben comunicarse. La comunicación (intercambio de valores) se realiza mediante canales. Este concepto evita que la información se pierda o sea sobreescrita, aunque genera una carga mayor de memoria.

3.1. Creación de una Goroutine

Para crear una “goroutine” solo basta con invocar una función anteponiendo la palabra reservada “go”:

```
1  func IsReady(what string, minutes int64) {
2     time.Sleep (minutes * 60 * 1e9)
3     fmt.Println (what, "está listo.")
4  }
5
6  go IsReady("Té", 6)
7
8  go IsReady("Café", 2)
9
10 fmt.Println("Estoy esperando...")
```

El ejemplo anterior imprimirá:

```
Estoy esperando... (De forma inmediata)
Café está listo    (2 minutos después)
Té está listo      (6 minutos después)
```

3.2. Canales

A menos que dos goroutines puedan comunicarse, estas no pueden coordinarse o sincronizarse de ninguna forma. Por ello **Go** tiene un tipo denominado channel que provee comunicación y sincronización. También posee estructuras de control especiales para los canales que facilitan la programación concurrente.

3.2.1. El tipo channel

En su forma más simple, el tipo se declara como:

```
1  chan tipo_elemento
```

Dada una variable de ese estilo, puedes enviar y recibir elementos del tipo tipo_elemento.

Los canales son un tipo de referencia, lo que implica que se puede asignar una variable “chan” a otra y que ambas variables accedan al mismo canal.

```
1  var c = make(chan int)
```

3.2.2. El operador <-

El operador “<−” es un operador que nos permite pasar datos a los canales. La flecha apunta en la dirección en que se pretende que sea el flujo de datos.

Como operador binario, “<−” envía a un canal:

```
1  var c chan int
2  c <- 1 // Envía 1 al canal c
```

Como un operador unario prefijo, “<−" recibe datos de un canal:

```
1  v = <- c // Recibe un valor de c y se lo asigna a v.
2  <- c // Recibe un valor de c, descartándolo
3  i := <- c // Recibe un valor de c inicializando i
```

3.2.3. Semántica de canales

Por defecto la comunicación en **Go** se realiza de forma sincrónica. Esto quiere decir:

- Una operación “send” en un canal bloquea la ejecución hasta que se ejecuta una operación “receive” en el mismo canal.

- Una operación “receive” en un canal bloquea la ejecución hasta que una operación “send” es ejecutada en el mismo canal.

De esta forma, la comunicación es además una forma de sincronización: 2 goroutines intercambiando datos a través de un canal, se sincronizan en el momento en que la comunicación es efectiva.

3.2.4. Ejemplo de comunicación

Para ejemplificar la sintaxis de canales:

```
1  func pump(ch chan int) {
2     for i := 0; ; i++ {
3        ch <- i
4     }
5  }
6
7  ch1 := make(chan int)
8  go pump (ch1) // Se bloquea; ejecutamos:
9
10 fmt.Println (<-ch1) // Imprime 0
11
12 func suck (ch chan int) {
13    for {
14       fmt.Println (<-ch)
15    }
16 }
17
18 go suck (ch1) // Miles de números se imprimen
19
20 // Se puede colar la función principal y pedir un valor
21 fmt.Println(<-ch) // Imprime un valor cualquiera (314159)
```

3.2.5. Direccionalidad de un canal

En su forma más simple, una variable de tipo canal es una variable sin memoria (no posee un buffer), es síncrona y que puede ser utilizada para enviar y recibir.

Una variable de tipo canal puede ser anotada para especificar que únicamente puede enviar o recibir:

```
1  var recv_only <-chan int
2  var send_only chan <- int
```

Todos los canales son creados de forma bidireccional, pero podemos asignarlos a variables de canales direccionales. Esto es útil por ejemplo en las funciones, para tener seguridad en los tipos de los parámetros pasados:

```
1  func sink(ch <- chan int) {
```

```
2     for { <- ch }
3  }
4
5  func source(ch chan<- int) {
6     for { ch <- 1 }
7  }
8
9  var c = make(chan int) // Bidireccional
10 go source(c)
11 go sink(c)
```

3.2.6. Canales síncronos

Los canales síncronos no disponen de un buffer. Las operaciones “send” no terminan hasta que un “receive” es ejecutado. Veamos un ejemplo de un canal síncrono:

```
1  c := make(chan int)
2
3  go func () {
4     time.Sleep (60 * 1e9)
5     x := <- c
6     fmt.Println("Recibido", x)
7  }
8
9  fmt.Println("Enviando", 10)
10 c <- 10
11 fmt.Println("Enviado", 10)
```

Y la salida sería:

```
1  Enviando 10  (Ocurre inmediatamente)
2  Enviado 10   (60s después, estas dos lineas aparecen)
3  Recibido 10
```

3.2.7. Canales asíncronos

Un canal asíncrono con buffer puede ser creado pasando a make un argumento, que será el número de elementos que tendrá el buffer.

```
1  c := make (chan int, 50)
2
3  go func () {
4     time.Sleep (60 * 1e9)
5     x := <-c
6     fmt.Println ("recibido", x)
7  }
```

```
 8
 9  fmt.Println ("Enviando", 10)
10  c <- 10
11  fmt.Println ("enviado", 10)

    enviando 10  (Ocurre inmediatamente)
    enviado 10   (Ahora)
    recibido 10  (60s después)
```

Nota.- El buffer no forma parte del tipo canal, sino que es parte de la variable de tipo canal que se crea.

3.2.8. Multiplexación

Los canales son valores de ”primer tipo”, es decir, pueden ser enviados a través de otros canales. Esta propiedad hace que sea fácil de escribir un servicio multiplexador dado que el cliente puede proporcionar, junto con la petición, el canal al que debe responder.

```
 1  chanOfChans := make(chan chan int)
 2
 3  O de forma mas típica
 4
 5  type Reply struct { ... }
 6
 7  type Request struct {
 8     arg1, arg2, arg3 some_type
 9     replyc chan *Reply
10  }
```

Ahora desarrollaremos un ejemplo de cliente-servidor.

3.2.8.1. El servidor

Implementación del servidor:

```
 1  type request struct {
 2     a, b int
 3     replyc chan int
 4  }
 5
 6  type binOp func (a, b int) int
 7
 8  func run(op binOp, req *request) {
 9     req.replyc <- op(req.a, req.b)
10  }
```

```
11
12 func server(op binOp, service chan *request) {
13    for {
14       req := <- service // Aquí se aceptan las peticiones
15       go run(op, req)
16    }
17 }
```

Para comenzar el servidor, lo hacemos de la siguiente forma:

```
1  func startServer (op binOp) chan *request {
2     req := make(chan *request)
3     go server(op, req)
4     return req
5  }
6
7  var adderChan = startServer(
8     func a, b int) int { return a + b }
9  )
```

3.2.8.2. El cliente

Implementación del cliente:

```
1  func (r *request) String() string {
2     return fmt.Sprintf("\%d+\%d=\%d", r.a, r.b, <-r.replyc)
3  }
4
5  req1 := &request{ 7, 8, make(chan int) }
6  req2 := &request{ 17, 18, make(chan int) }
7
8  adderChan <- req1
9  adderChan <- req2
10 fmt.Println (req2, req1)
```

4. Manejo de Memoria

Go no posee aritmética de punteros, según los creadores, por razones de seguridad. Sin aritmética de punteros no es posible obtener una dirección de memoria ilegal y que sea utilizada de forma incorrecta.

Por otro lado, la falta de aritmética de punteros simplifica la implementación del recolector de basura.

4.1. Recolección de Basura

Actualmente, el compilador de **Go** posee un Garbage Collector muy simple pero efectivo basado en el ”marcado de barrido” (mark and sweep), es decir, marca aquello que puede ser eliminado, y al ser activado, se borra.

Este recolector, se basa en los siguientes conceptos fundamentales:

- Stop the World: El recolector detiene la ejecución de la aplicación para realizar su trabajo. La frecuencia de estas pausas es proporcional al tamaño del heap.

- Non-Compacting: El recolector nunca mueve un objeto dentro de la memoria. Una vez que el objeto es creado en una dirección determinada, este persiste hasta que es liberado. Esta característica no permite compactar la memoria utilizada por una aplicación.

- Non-Generational: Otros recolectores separan los objetos en Jóvenes y Viejos. Pero el recolector de Go los considera a todos por igual, no diferencia entre objetos recién creados u objetos que llevan más tiempo ejecutándose.

- Conservativo: Un recolector de basura conservativo asume que cualquier dato puede ser un puntero. De esta forma el recolector realiza una búsqueda en memoria de un dato que pueda parecer un puntero y realiza una copia de dicho puntero en una zona de memoria donde impedir que sea reciclado. Este proceso puede producir un falso/positivo ya que el puntero puede apuntar a un objeto que ya fue borrado previamente, y de todas formas el puntero persiste.

5. AppEngine

AppEngine es una plataforma creada por Google, orientada a crear servicios web y distribuirlas en base a su infraestructura.

Al ser un sistema pensado para la nube, provee acceso a todos los servicios de Google mediante APIs (Datastore, Mail, Users, XMPP) y sobre todo provee un entorno muy seguro.

Las aplicaciones corren en un sandbox y se encuentran distribuidas en múltiples data-centers.

No es un framework, sinó que es un conjunto de frameworks, librerías y utilidades, con soporte para **Go**, Python y Java.

6. Compiladores

Existen dos compiladores de **Go**:

6g/8g/5g - Compilan a AMD64 / x86 / ARM respectivamente. Comandos:

- "go run ejemplo.go": Compila y ejecuta el programa
- "go build ejemplo.go": Compila el programa y dependencias y genera un ejecutable
- "go get paquete": Permite descargar paquetes y agregarlos al Runtime

gccgo - Frontend para GCC escrito en C++.

- Se utiliza de la misma forma que el compilador de C

Ambos compiladores son multiplataforma.

7. Ejemplo de uso

El ejemplo realizado para nuestra exposición, consistió en el desarrollo de un Blog utilizando AppEngine y **Go**.

Este sistema esta compuesto principalmente por un paquete "blog" el cual contiene todas las funciones y estructuras necesarias para el funcionamiento del mismo.

Analizando los archivos del proyecto, podemos encontrar una carpeta llamada "blog" la cual representa el paquete antes mencionado, y otra carpeta llamada "static" la cual contiene todos los archivos que son estáticos y no es necesario que sean interpretados por el servidor, este solo debe servirlos.

Si analizamos la carpeta contenedora del paquete "blog", podemos encontrar "admin.go" que contiene aquellas funciones encargadas de la administración del blog, "categorias.go" que define las funciones y estructuras necesarias para utilizar categorías, "entradas.go" es similar al archivo anterior pero en este se define todo lo necesario para procesar entradas; por último se encuentra el archivo "mostrarEntradas.go" que es el encargado de listar todas las entradas y categorías.

Este último archivo es el más importante dentro del funcionamiento del sistema, ya que en este se define la función "init", que según el estándar definido por AppEngine, esta es la función que primero se ejecuta dentro de una aplicación. Analizando esta función se puede ver que lo único que hace es asignar a la request "/" (enviada por el navegador) una función declarada dentro del paquete. Esta función es "mostrarEntradas", la cual recibe como parámetros la salida o writer (donde debe imprimir el sitio web) y un elemento "request" el cual contiene toda la información intercambiada entre el navegador y el servidor. Dentro del scope de la función se obtienen las entradas y las categorías, las cuales son guardadas en arrays y estos a su vez dentro de un “struct” llamado "contenido". Paso seguido, el programa ejecuta el template (que contiene el diseño del blog) reemplazando cada sector del diseño con los datos almacenados en el struct.

El contenido de los archivos "entradas.go" y "categorias.go" es muy similar, y se encargan de obtener los datos almacenados en la DataStore (base de datos de AppEngine) para luego poder ser pegados dentro del diseño del blog.

Por último, "admin.go" contiene las funciones encargadas de la administración de entradas y categorías, ya sea la alta-baja-modificación de los datos almacenados en DataStore. En este archivo también se define la función "init" ya que el administrador enviará ciertas request para administrar el sitio, siendo estas diferentes a las request enviadas por un usuario normal.

Al utilizar las metodologías propuestas por AppEngine, cada "request" o "pedido" enviado por el usuario, puede ser asignado a una función diferente, y de esta forma nos permite descentralizar las características del sistema, sin depender unas de otras.

Para acceder al código fuente del sistema, se puede clonar el repositorio git:

http://code.google.com/p/amouly-codes/source/browse/#git%2FAppEngine%2Famblog

8. Para qué sirve

Go es muy eficiente a la hora de realizar aplicaciones modernas, orientadas a los servicios web, donde existe un ambiente de constante cambio.

Sirve para crear soluciones en poco tiempo, ya que su sintaxis busca justamente eso.

No sirve para programar aplicaciones de bajo nivel, ya que no provee las estructuras necesarias.