UNPSJB

LICENCIATURA EN SISTEMAS OPGCPI

PARADIGMAS DE LENGUAJES Y PROGRAMACIÓN

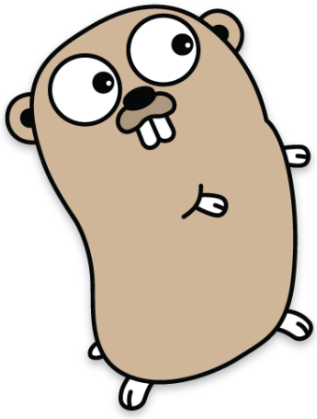
Trabajo de investigación

Golang

Cátedra
Lic. Romina Stickar
Lic. Lautaro Pecile

Integrantes:
Luciano Serruya Aloisi

5 de julio de 2018



Índice

1. Introducción **2**

2. Criterios de evaluación **3**

3. Sistema de tipos **4**
3.1. Tipos de datos 5

4. Estructuras de control **6**

5. Paradigmas **7**

6. Manejo de eventos inusuales **10**

7. Concurrencia **12**
7.1. `goroutines` 13
7.2. Canales 13

8. Conclusión **15**

1. Introducción

En el año 2007, tres ingenieros de Google (*Robert Griesemer*, *Rob Pike*, y *Ken Thompson*) comenzaron a diseñar el lenguaje de programación *Go*, como proyecto secundario. Para el 2008, empezaron con el desarrollo de un compilador y un *runtime* [1]. El 30 de Octubre de 2009, Rob Pike dio la primer charla sobre Go en una *Google Techtalk* [12], pero recién para el 10 de Noviembre de ese mismo año el proyecto fue oficialmente anunciado. Debido a que el proyecto es *open-source*, se formó una gran comunidad que aceleró el desarrollo y uso del lenguaje

Desde Mayo de 2010, Google utiliza Go en producción para la infraestructura de sus servidores (lo cual demuestra que la empresa apuesta en él, y que el lenguaje tiene peso como para estar en producción [1])

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

"Hola mundo" en Go

Sus creadores decidieron encarar el proyecto al ver que no había surgido ningún lenguaje de programación (de bajo nivel pero con amplios niveles de abstracción) que sea adecuado para el panorama computacional de hoy en día. Por lo tanto, se considera a Go como "el C del siglo 21" [11]

El lenguaje se utiliza mucho para construir servidores, herramientas y sistemas para programadores, pero no deja de ser un lenguaje de propósito general.

Existen muchos lenguajes de programación que influenciaron distintas decisiones de diseño de Go, siendo alguno de ellos C (sintaxis, sentencias de flujo de control, tipos de datos básicos, **pasaje de parámetros por copia**, punteros), pero también Pascal, Modula-2 (sistema de paquetes), y CSP (manejo de concurrencia)

[1] Software diseñado para soportar la ejecución de programas escritos en algún lenguaje de programación [15]

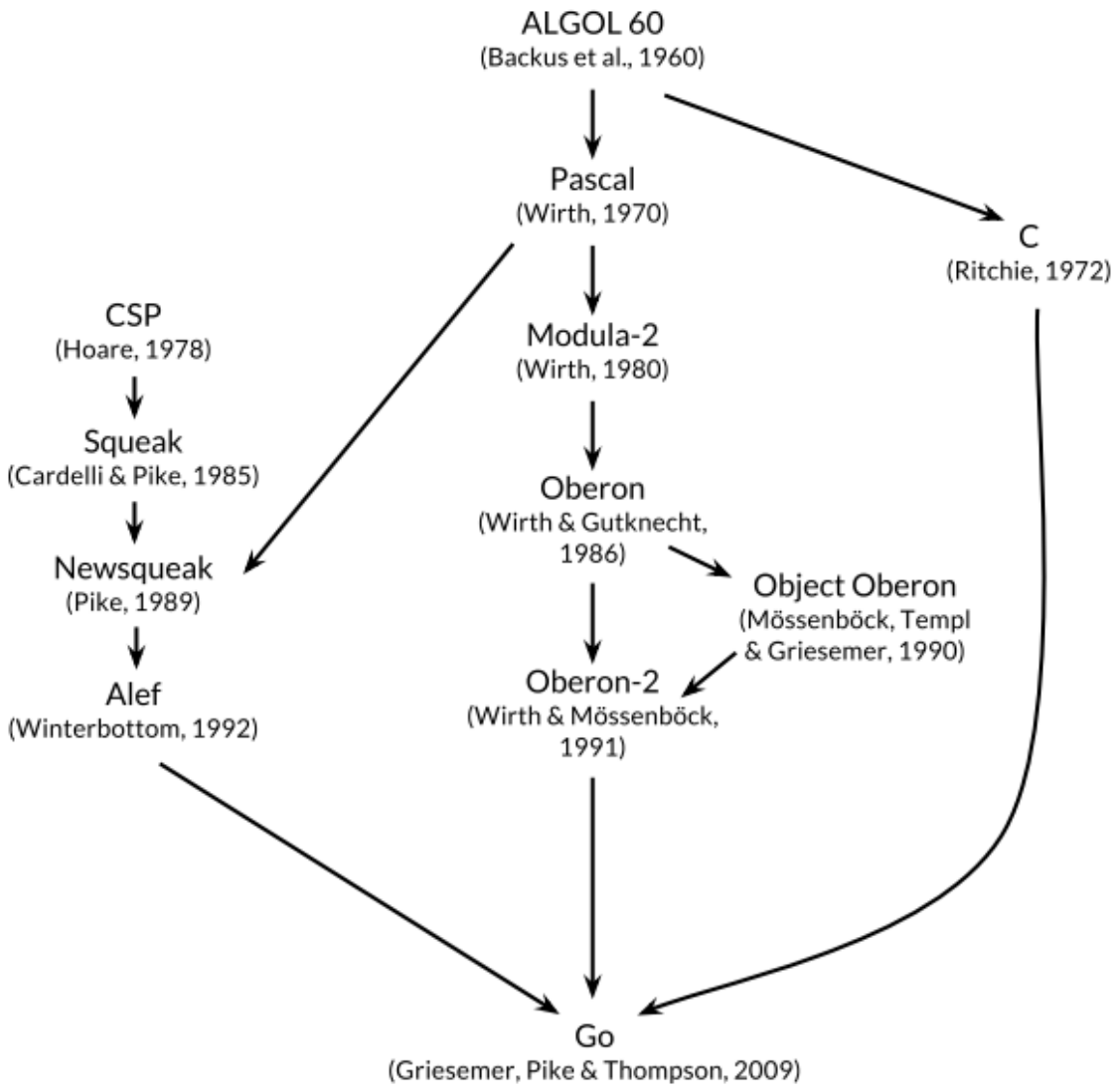


Figura 1: Ancestros de Go [11]

2. Criterios de evaluación

Go tiene una sintaxis consisa y regular, conseguida con pocas palabras reservadas. Esto aumenta la velocidad de compilación (las sentencias pueden ser evaluadas por la gramática sin una *tabla de símbolos*), y reduce las líneas de código; una especificación completa del lenguaje puede hallarse en [9]. También intenta establecer un o dos formas para realizar una determinada tarea, lo cual logra que **facilita la lectura del código**.

El lenguaje es lo suficientemente inteligente como para *inferir tipos* en asignaciones

con el operador ":=". Esta característica favorece a la **escritura** del código.

```
1 s := ""
2 var s string
3 var s = ""
4 var s string = ""
```

Distintas formas de declarar una variable

El diseño y la implementación del lenguaje favorecen más que nada a la **seguridad** y al **costo de ejecución** de los programas. Para ello utiliza un *recolector de basura* (*Garbage Collector*), que se encarga de administrar el uso de la memoria para evitar *referencias colgantes* o *memory leaks* (se podría considerar también como un incremento en la facilidad de escritura)

Go ha reemplazado lenguajes dinámicos porque balancea *expresividad* con *seguridad*; los programas escritos en Go típicamente corren más rápido que programas escritos en lenguajes dinámicos y sufren muchos menos errores en tiempo de ejecución por problemas de tipos inesperados [11]. La seguridad que provee el lenguaje se da gracias a su **sistema de tipos** y a que es ***memory-safe*** (manejo de memoria seguro) [12].

3. Sistema de tipos

Go es un lenguaje **fuerte y estáticamente tipado**. Todas las expresiones tienen un tipo determinado en tiempo de compilación, y una vez declarada una variable no puede cambiar su tipo.

```
1 aString := "Hello world" // Variable de tipo cadena
2
3 aString := 3 // No se puede redefinir la variable
4 aString := "Bye" // No se puede redefinir la variable
5 aString = 3 // Asignación ilegal
6 aString = "Bye world" // Asignación válida
```

Ejemplos de asignaciones válidas e inválidas (nótese el uso de los operadores ":=" y "=")

No provee conversiones implícitas, por lo cual no es posible tener **expresiones mixtas** [2]. Tampoco tiene constructores o destructores, pero cuando se crea una variable se la

[2] expresiones que involucren más de un tipo de dato

inicializa con el **valor cero** o **valor inicial** de su tipo de dato.

3.1. Tipos de datos

Go provee distintos tipos de datos simples por defecto, ellos son:

- Numéricos: su valor inicial es 0
  - Enteros: con y sin signo; también brinda opciones dependientes de la plataforma
  - Punto flotante: 32 o 64 bits
- Cadenas: se inicializan a cadena vacía (“”)
- Booleanos: se inicializan a `false`

Tipos de datos complejos provistos por el lenguaje son los **arreglos** (listas indexables de un tipo de dato específico con un tamaño predeterminado), las **rebanadas** o **slices** (segmento de un arreglo, su tamaño puede variar en tiempo de ejecución), y los **mapas** (listas asociativas o *clave-valor*, se definen con un tipo de dato para la clave y otro para el valor). Cabe aclarar que el tamaño de un arreglo forma parte de la definición de su tipo, por lo tanto no se puede asignar a una variable cuyo tipo sea un arreglo de tamaño `X` un arreglo de tamaño distinto de `X`.

También existe la posibilidad de definir **estructuras** o **registros**, cuyos campos serán de un tipo de dato simple o complejo.

```
1 type Circle struct {
2     x, y, r float64
3 }
```

Definición de un tipo de registro `Circle` con tres campos del mismo tipo

Los tipos de datos complejos se inicializan con los valores iniciales que tenga cada uno de los tipos de datos que los componen.

Otro tipo de dato complejo que provee Go es el tipo de dato **puntero**. Se define en base a otro tipo de dato, y su valor será el de una **dirección** de memoria que almacenará un valor del tipo de dato definido para el puntero.

Al igual que C, se obtiene el valor apuntado por el puntero mediante el operador $*$ (*dereferenciamiento*), y para conseguir la dirección de un tipo de dato se utiliza el operador `&`

4. Estructuras de control

Go cuenta con tres estructuras de control, dos de selección y una de iteración.

La única estructura de control que implementa es el ciclo `for`, con la siguiente sintaxis:

```
for <INICIALIZACIÓN>; <CONDICIÓN>; <INCREMENTO> {
    // Cero o más sentencias
}
```

Sintaxis del ciclo `for` (la llave debe estar en la misma línea que las sentencias de incremento)

Con esta única estructura se puede simular el ciclo `while` (solamente especificando la condición, y controlándola dentro del bloque de sentencias con las palabras reservadas `return`, `continue`, y `break`), o un ciclo infinito (`for` sin ninguna indicación)

Si se desea iterar sobre una estructura indexable (arreglo o mapa), la claúsula `range` devuelve una clave y su respectivo valor

```
for i, value := range arr {
    fmt.Printf("POSICION %d : VALOR %v\n", i, value)
}
```

Iterando sobre un arreglo con `range`

La estructura de decisión más simple con la que cuenta el lenguaje es el `if`. Al igual que el `for`, la condición no necesariamente se escribe entre paréntesis, y las llaves que indican el bloque de instrucciones son necesarias y van en la misma línea que la condición. Se puede indicar también, con la palabra reservada `else`, el bloque de sentencias a ejecutar si no se cumple la condición especificada por el `if`.

```
for i := 0; i < 10; i++ {
    if i % 2 == 0 {
        fmt.Printf("El valor %d es par\n", i)
    } else {
        fmt.Printf("El valor %d es impar\n", i)
    }
}
```

Ejemplo de un `for`, un `if..else`, y operaciones aritméticas

Otra estructura de decisión es el `switch`; consiste en indicar una *sentencia simple* [3], dentro de las llaves los distintos casos posibles con la palabra `case` y seguido de dos puntos (:) el bloque de sentencias que se desea ejecutar si se cumple esa condición.

El comportamiento por defecto del `switch` es distinto al `switch` de C, en tanto que si se cumple una condición, no seguirá automáticamente ejecutando el bloque de sentencias de la condición que esté por debajo de la que se cumplió (si se desea tal comportamiento, se lo puede especificar con la palabra `fallthrough` [2]).

Por último, se puede especificar un bloque de sentencias a ejecutar si ninguna de las condiciones anteriores se cumplieron; tal bloque se indica con la palabra reservada `default`

```
1 switch var1 {
2     case val1:
3         ...
4     case val2:
5         ...
6     default:
7         ...
8 }
```

Estructura de decisión múltiple `switch`

5. Paradigmas

Principalmente, Go es un lenguaje de programación **procedural**. Utiliza estructuras de control, asignación de variables, constantes, y funciones que manipulan datos.

Las funciones pueden recibir cero o más parámetros de cualquier tipo, y pueden devolver o más resultados (la expresión `range` devuelve dos valores, el índice de una estructura iterable y su correspondiente valor). La cantidad de parámetros que recibe una función puede ser variable, especificándolo con `...` antes del tipo del parámetro.

El pasaje de parámetros es siempre por **copia**, ya sea el valor actual de una variable, o la *dirección* de dicha variable. Con esta última opción, la función receptora de la *referencia* podría alterar el valor de la variable y ese cambio persistiría después de finalizar la función, mientras que con la primer opción no sería posible.

[3] Sentencia que devuelva un valor

```
1 func sumatoria(args ...int) int{
2     total := 0
3     for _, value := range args {
4         total += value
5     }
6     return total
7 }
```

Función con cantidad variable de argumentos

La firma de una función siempre empieza con la palabra reservada `func`, le sigue el nombre de la función, luego entre paréntesis los argumentos que recibe, indicando primero su nombre y después su tipo (si es que recibe parámetros, si no recibe se dejan los paréntesis vacíos o se pueden obviar), y por último el resultado que devuelve (si son varios, se especifica entre paréntesis y separado por comas el tipo de dato de cada valor devuelto).

Go considera las funciones como *ciudadanos de primera clase* [14], por lo tanto pueden ser recibidas como parámetros (una función puede recibir una función), o devueltas como resultado (una función que devuelve una función[4])

El lenguaje también permite definir *funciones anónimas*; su definición se trata como una expresión. Se pueden almacenar en variables, y no llevan nombre.

Antes del nombre de una función, se puede especificar *el tipo de dato receptor*, lo cual convertiría a la función en un *método*. De esta forma, se pueden agregar comportamientos a los tipos de datos definidos por el usuario (aunque puede ser casi cualquier tipo de dato [3]). Entonces, la combinación entre una estructura (`struct`) y sus métodos es el equivalente en Go a una **clase del paradigma orientado a objetos**.

```
1 func (persona *Persona) Saludar() string {
2     return fmt.Sprintf("Hola! Mi nombre es %s",
3         persona.nombre)
4 }
```

Método *Saludar* para una persona (nótese que el método recibe un puntero a una persona, Go automáticamente lo dereferencia)

Go también permite definir registros con la siguiente estructura:

[4] El patrón de diseño *Decorador* podría ser implementado gracias a esta característica

```go
type Persona struct {
    nombre string
    edad int
}

type Alumno struct {
    Persona
    notas []int
}

func main() {
    a := Alumno{
            Persona{"Luciano", 21},
            []int{7,8,9},
        }
    fmt.Printf("Alumno: %s", a.nombre)
}
```

Definición de un registro con registros embebidos (anónimo)

El ejemplo anterior muestra la única forma que provee Go para simular una **herencia** de clases. Sin embargo, no se trataría de una herencia de clases *per se*, sino más bien de una **composición** de registros.

Por lo tanto, para crear estructuras de datos complejas, la forma de hacerlo en Go es a través de la composición, y no de la herencia.

El lenguaje permite definir **interfases**, que son conjuntos de métodos que un tipo debe implementar. A diferencia de lenguajes como Java, el tipo de dato que implemente la interfase no tiene que indicar qué interfase implementa, sino que directamente debe implementar los **métodos** que ésta define.

> *Las interfases en Go proveen una manera de* especificar el comportamiento *de un objeto: si algo puede hacer* esto*, entonces se puede usar* aquí *[4]*

Supóngase que se definió una interfase `Saludador` con el método `Saludar() string` (no recibe nada, y devuelve una cadena), se podría crear la siguiente función que recibe una cantidad variable de argumentos que implementen dicha interfase

```go
func Saluden(saludadores ...Saludador) () {
    for i, obj := range saludadores {
        fmt.Printf("[%d] %s\n", i, obj.Saludar())
    }
}
```

Comportamiento polimórfico

De esta forma Go implementa el **polimorfismo**. No es polimorfismo de clase (ya que Go no maneja el concepto de clase), sino de **comportamiento** (no importa de qué tipo sea, lo único que importa es qué se le puede pedir o qué comportamiento tiene)

6. Manejo de eventos inusuales

La forma idiomática para manejar errores es **devolviendo valores de error**. Muchas funciones de su librería estándar utilizan esta metodología para indicar que no se pudo completar la tarea satisfactoriamente. La posibilidad de que las funciones retornen más de un valor facilita la implementación de esta técnica.

```go
file, err := os.Open("test.txt")
if err != nil {
    // manejar el error
    return
}
```

La función `Open` de la librería `os` devuelve el descriptor del archivo que se quiere abrir, y un valor de error en caso de que no se haya podido

Por otro lado, Go no maneja el concepto de *excepciones*, pero sí tiene dos funciones que emulan un comportamiento muy similar: `panic` y `recover`.

La función `panic` recibe un parámetro que sirve como mensaje de error y no devuelve nada. Pone al hilo de ejecución en un estado de error, que se irá propagando hasta ser manejado, o bien hasta finalizar la *pila de llamadas* y así mismo la ejecución del programa.

La función `recover` es la encargada de recuperar el control del hilo de ejecución. No recibe nada y devuelve el valor que recibió `panic`. Una llamada a `recover` durante una

ejecución normal no tendrá ningún efecto; solamente es útil dentro de *funciones diferidas* [7].

Una **función diferida** se especifica con la palabra reservada `defer` antes de su invocación, y consiste en retrasar su ejecución hasta terminada la unidad llamadora. Son útiles para cerrar archivos o conexiones con bases de datos, liberar recursos, o manejar errores.

Cuando el hilo de ejecución entra en pánico, se detiene el flujo normal de ejecución, se llevan a cabo todas las funciones diferidas que tenga la unidad, y se regresa al contexto que la invocó. Si una unidad tiene más de una función diferida, se ejecutan en orden *LIFO* (*Last In, First Out, Último en Entrar, Primero en Salir*).

```go
package main

import "fmt"

func main() {
    panicAndRecover()
    fmt.Println("Sentencia muy importante")
}

func panicAndRecover() {
    defer func() {
        if err := recover(); err != nil {
            fmt.Printf("[ERROR] %v\n", err)
        }
    }()
    /*
    Acá hacer algo que termine en
    una situación que no pueda
    manejar esta función
    */
    panic("Un error que no puedo manejar D:")
}

/*
SALIDA:
    [ERROR] Un error que no puedo manejar D:
    Sentencia muy importante
*/
```

Defer, Panic, Recover [10]

7. Concurrencia

Una de las principales y más novedosas características de Go es su soporte nativo para la concurrencia y paralelismo. Provee herramientas para ejecutar distintas porciones de

código de forma concurrente, y también mecanismos para comunicarlos, bloquearlos, y sincronizarlos.

7.1. goroutines

Go ejecuta código concurrentemente a través de las goroutines. No existe ninguna asociación directa en una goroutine y un hilo del sistema operativo: puede ser multiplexada a uno o más hilos, dependiendo de su disponibilidad; esto se logra gracias al *planificador de goroutines* y al *runtime* de Go [5]

Para correr código *en una rutina distinta de la que está ejecutando el hilo principal*, simplemente se debe anteponer la palabra reservada go a la llamada a la función deseada.

```
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func hello() {
8      fmt.Println("Hello world goroutine")
9  }
10 func main() {
11     go hello()
12     fmt.Println("main function")
13 }
```

Ejecución concurrente con goroutines [13]

7.2. Canales

Uno de los principales problemas que se presenta en la programación concurrente es la **comunicación entre los distintos procesos en ejecución**. Para resolver este problema, Go provee de un mecanismo de comunicación entre las goroutines que son los **canales**.

Los canales proveen un mecanismo para que dos goroutines se comuniquen entre sí y sincronicen sus ejecuciones [8]. Se definen con un tipo de dato, que será el tipo de dato de los valores que viajarán a través de él.

```go
package main
import (
    "fmt"
    "time"
)
func pinger(c chan string) {
    for i := 0; ; i++ {
        // Enviamos una cadena por el canal
        c <- "ping"
    }
}
func printer(c chan string) {
    for {
        // Recibimos una cadena por el canal
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}
func main() {
    var c chan string = make(chan string)
    go pinger(c)
    go printer(c)
    var input string
    fmt.Scanln(&input)
}
```

Dos `goroutines` que se comunican a través de un canal de cadenas [8]

Las acciones sobre un canal son **bloqueantes**, lo que significa que la ejecución se detendrá en ese punto hasta que la pueda completar exitosamente.

Si una `goroutine` intenta leer de un canal del cual no hay nada para leer (está vacío, no hay valores nuevo que leer), se bloqueará hasta que otra `goroutine` escriba algo en el canal.

Análogamente, sucedería lo mismo cuando se quiera escribir y el canal esté lleno (en el caso de los canales con buffer, los cuales tienen un tamaño predeterminado).

8. Conclusión

Go es un lenguaje moderno, pensado para grandes sistemas, soporta tanto el paradigma procedural como el orientado a objetos (con una perspectiva muy distinta a la de otros lenguajes que soportan el paradigma), y está en crecimiento constante.

Fue diseñado con la idea en mente de que las computadoras de hoy en idea tienen múltiples núcleos y capacidad de correr tareas en forma concurrente y paralela, por lo tanto la facilidad que brinda para una programación que aproveche dichas características.

Permite construir estructuras de datos complejas y armar grandes niveles de abstracción a través de la composición y polimorfismo de comportamiento. Sin embargo no deja de tener un rendimiento comparable con el de C o C++ [6]

Referencias

[1] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 1. Origins and Evolution.

[2] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 5. Control Structures.

[3] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 10. Structs and Methods.

[4] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 11. Interfaces and reflection.

[5] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 14. Goroutines and channels.

[6] Ivo Balbaert. «The Way to Go: A Thorough Introduction to the Go Programming Language». En: iUniverse, 2012. Cap. 3. Editors, IDE's and Other tools.

[7] The Go Blog. *Defer, Panic, Recover*. Ago. de 2010. URL: https://blog.golang.org/defer-panic-and-recover.

[8] Caleb Doxsey. «An Introduction to Programming in Go». En: Google, 2012. Cap. 10. Concurrency.

[9] Golang. *Especificaciones del lenguaje Go*. Mayo de 2018. URL: https://golang.org/ref/spec.

[10] Saddam H. *go-101/ Defer, panic and recover*. Mar. de 2017. URL: https://medium.com/@thedevsaddam/go-101-defer-panic-and-recover-65a40ee7dcb4.

[11] Alan A. A. Donovan; Brian W. Kernighan. «The Go Programming Language». En: Addison-Wesley, 2016. Cap. Preface.

[12] Rob Pike. «The Go Programming Language». Google Techtalk. Oct. de 2009.

[13] Naveen Ramanathan. *Part 21: Goroutines*. Jul. de 2017. URL: https://golangbot.com/goroutines/.

[14] Wikipedia. *First-class citizen*. Mayo de 2018. URL: https://en.wikipedia.org/wiki/First-class_citizen.

[15] Wikipedia. *Runtime system*. Jun. de 2018. URL: https://en.wikipedia.org/wiki/Runtime_system.