

Seminario de Lenguajes opción Go

Raúl Champredonde

Seminario de Lenguajes opción Go

- Concurrency
- Goroutines
- WaitGroup
- Channels
- Select
- Monitores
- Semáforos

Concurrencia

- Concurrencia
- Paralelismo
- Threads / Task / Process
- Goroutine: función que es capaz de ejecutar concurrentemente con otras funciones

Concurrencia - Goroutines

```
func f(n int) {  
    for i := 0; i < 10; i++ {  
        fmt.Println(n, ":", i)  
    }  
}  
  
func main() {  
    go f(0)  
    fmt.Scanln()  
}
```

```
0 : 0  
0 : 1  
0 : 2  
0 : 3  
0 : 4  
0 : 5  
0 : 6  
0 : 7  
0 : 8  
0 : 9
```

Concurrencia - Goroutines

```
func f(n int) {  
    for i := 0; i < 10; i++ {  
        fmt.Println(n, ":", i)  
    }  
}  
  
func main() {  
    for i := 0; i < 10; i++ {  
        go f(i)  
    }  
    fmt.Scanln()  
}
```

2 : 0
2 : 1
2 : 2
2 : 3
2 : 4
2 : 5
2 : 6
2 : 7
2 : 8
2 : 9
0 : 0
0 : 1
0 : 2
0 : 3
0 : 4
0 : 5
0 : 6
0 : 7
0 : 8
0 : 9

3 : 0
3 : 1
3 : 2
3 : 3
3 : 4
3 : 5
3 : 6
3 : 7
3 : 8
3 : 9
6 : 0
6 : 1
6 : 2
6 : 3
6 : 4
4 : 0
4 : 1
8 : 0
8 : 1
8 : 2

8 : 3
8 : 4
8 : 5
8 : 6
8 : 7
8 : 8
8 : 9
6 : 5
6 : 6
6 : 7
6 : 8
6 : 9
5 : 0
5 : 1
5 : 2
5 : 3
5 : 4
5 : 5
5 : 6
5 : 7

5 : 8
5 : 9
4 : 2
4 : 3
4 : 4
4 : 5
4 : 6
4 : 7
4 : 8
4 : 9
1 : 0
1 : 1
1 : 2
1 : 3
1 : 4
1 : 5
1 : 6
1 : 7
1 : 8
1 : 9

9 : 0
9 : 1
9 : 2
9 : 3
9 : 4
9 : 5
9 : 6
9 : 7
9 : 8
9 : 9
7 : 0
7 : 1
7 : 2
7 : 3
7 : 4
7 : 5
7 : 6
7 : 7
7 : 8
7 : 9

Concurrencia - Goroutines

```
import ("fmt"; "math/rand"; "time")

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    fmt.Scanln()
}
```

2 : 0	3 : 2	1 : 2	5 : 6	4 : 6
1 : 0	9 : 2	0 : 3	3 : 6	7 : 9
0 : 0	0 : 1	8 : 3	6 : 7	8 : 8
6 : 0	6 : 2	1 : 3	4 : 5	9 : 6
4 : 0	5 : 2	7 : 4	8 : 5	0 : 6
8 : 0	3 : 3	2 : 4	9 : 5	3 : 9
7 : 0	5 : 3	4 : 4	0 : 5	5 : 8
9 : 0	5 : 4	1 : 4	7 : 7	5 : 9
5 : 0	4 : 2	3 : 5	3 : 7	9 : 7
3 : 0	0 : 2	5 : 5	2 : 8	1 : 8
5 : 1	4 : 3	2 : 5	8 : 6	4 : 7
3 : 1	7 : 2	6 : 6	1 : 6	0 : 7
7 : 1	3 : 4	8 : 4	3 : 8	4 : 8
6 : 1	6 : 3	1 : 5	6 : 8	8 : 9
9 : 1	6 : 4	2 : 6	5 : 7	4 : 9
4 : 1	8 : 2	7 : 5	8 : 7	1 : 9
2 : 1	7 : 3	0 : 4	6 : 9	9 : 8
8 : 1	6 : 5	7 : 6	1 : 7	0 : 8
1 : 1	9 : 3	2 : 7	7 : 8	9 : 9
2 : 2	2 : 3	9 : 4	2 : 9	0 : 9

Concurrencia - Goroutines

```
package main
```

```
import (  
    "fmt"  
    "io"  
    "log"  
    "net/http"  
)
```

```
func main() {  
    go  
    responseSize("https://www.golangprograms.com")  
    go responseSize("https://coderwall.com")  
    go responseSize("https://stackoverflow.com")  
    go responseSize("https://web.arba.gov.ar")  
    fmt.Scanln()  
}
```

```
func responseSize(url string) {  
    fmt.Println("Getting ", url)  
    response, err := http.Get(url)  
    if err != nil {  
        log.Fatal(err)  
    }
```

```
    defer response.Body.Close()
```

```
    body, err :=  
    io.ReadAll(response.Body)  
    if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Println(url, len(body))  
}
```

```
Getting https://coderwall.com  
Getting https://stackoverflow.com  
Getting  
https://www.golangprograms.com  
Getting https://www.info.unlp.edu.ar  
https://www.info.unlp.edu.ar 184387  
https://stackoverflow.com 173099  
https://www.golangprograms.com 32693  
https://coderwall.com 185287
```

Concurrencia - Goroutines

```
package main

import (
    "fmt"
    "io"
    "log"
    "net/http"
)

var urls = []string{
    "https://www.golangprograms.com",
    "https://coderwall.com",
    "https://stackoverflow.com",
    "https://www.info.unlp.edu.ar",
}

func main() {
    for _, url := range urls {
        go responseSize(url)
    }
    fmt.Scanln()
}
```

```
func responseSize(url string) {
    fmt.Println("Getting ", url)
    response, err := http.Get(url)
    if err != nil {
        log.Fatal(err)
    }

    defer response.Body.Close()

    body, err :=
io.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(url, len(body))
}
```

```
Getting https://coderwall.com
Getting https://stackoverflow.com
Getting
https://www.golangprograms.com
Getting https://www.info.unlp.edu.ar
https://www.info.unlp.edu.ar 184387
https://stackoverflow.com 173099
https://www.golangprograms.com 32693
https://coderwall.com 185287
```


Concurrencia - WaitGroup

- WaitGroup
 - Permite que una goroutine espere la terminación de otras goroutines
 - El tipo `sync.WaitGroup` se puede pensar como un contador
 - El tipo `sync.WaitGroup` define los métodos:
 - `Add(delta 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

Concurrencia - WaitGroup

```
import (
    "fmt"
    "io"
    "log"
    "net/http"
    "sync"
)

var wg sync.WaitGroup

var urls = []string{
    "https://www.golangprograms.com",
    "https://coderwall.com",
    "https://stackoverflow.com",
    "https://www.info.unlp.edu.ar",
}

func main() {
    for _, url := range urls {
        wg.Add(1)
        go responseSize(url)
    }
    wg.Wait()
}
```

```
func responseSize(url string) {
    defer wg.Done()

    fmt.Println("Getting ", url)
    response, err := http.Get(url)
    if err != nil {
        log.Fatal(err)
    }

    defer response.Body.Close()

    body, err :=
io.ReadAll(response.Body)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(url, len(body))
}
```

```
Getting https://coderwall.com
Getting https://stackoverflow.com
Getting
https://www.golangprograms.com
Getting https://www.info.unlp.edu.ar
https://www.info.unlp.edu.ar 184387
https://stackoverflow.com 173099
https://www.golangprograms.com 32693
https://coderwall.com 185287
```

Concurrencia - WaitGroup

```
import (  
    ...  
)  
  
var wg sync.WaitGroup  
  
var urls = []string{  
    ...  
}  
  
func main() {  
    var wg sync.WaitGroup  
  
    for _, url := range urls  
    {  
        wg.Add(1)  
        go func(url string) {  
            defer wg.Done()  
            responseSize(url)  
        }(url)  
    }  
    wg.Wait()  
}
```

```
func responseSize(url string) {  
    defer wg.Done()  
  
    fmt.Println("Getting ", url)  
    response, err := http.Get(url)  
    if err != nil {  
        log.Fatal(err)  
    }  
  
    defer response.Body.Close()  
  
    body, err :=  
io.ReadAll(response.Body)  
    if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Println(url, len(body))  
}
```

```
Getting https://coderwall.com  
Getting https://stackoverflow.com  
Getting  
https://www.golangprograms.com  
Getting https://www.info.unlp.edu.ar  
https://www.info.unlp.edu.ar 184387  
https://stackoverflow.com 173099  
https://www.golangprograms.com 32693  
https://coderwall.com 185287
```

Concurrencia - Channels

- Channels
 - Mecanismo que permite que las goroutines se comuniquen y se sincronicen
 - Conducto “tipado” a través del cual una goroutine envía datos a otra
 - Por defecto, tanto la acción de enviar como la recibir bloquean a la goroutine que la ejecuta hasta que la del “otro extremo” esté lista.

Concurrencia - Channels

- Se declaran antes de usarlos

```
msg := make(chan string) | var msg chan string = make(chan string)
nums := make(chan int)    | var nums chan int = make(chan int)
```

- El “zero value” de un channel es `nil`

```
var nums chan int // nil
```

- Send

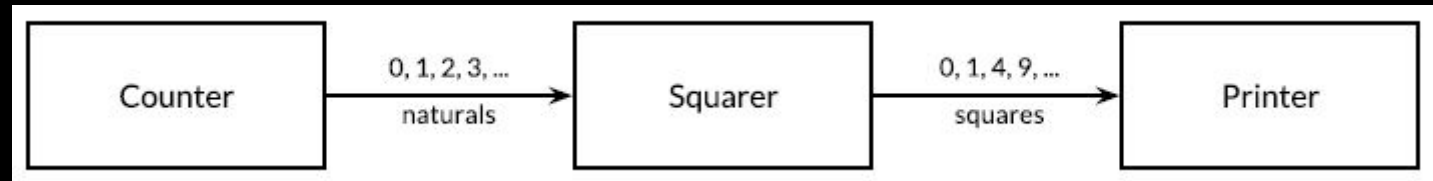
```
nums <- x
```

- Receive

```
x = <-nums
<-nums
```

Concurrencia - Channels

▪ Ejemplo



```
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    // Counter
    go func() {
        for x := 0; ; x++ {
            naturals <- x
        }
    }()
}
```

```
// Squarer
go func() {
    for {
        x := <-naturals
        squares <- x * x
    }
}()

// Printer
for {
    fmt.Println(<-squares)
}
}
```

```
0
1
4
9
16
25
36
49
64
81
100
121
144
...
```

Concurrencia - Channels

- Se pueden cerrar

```
close(nums)
```

- El receptor ...

```
x, ok := <-nums
```

si `ok` es `false` el channel no tiene más valores y está cerrado

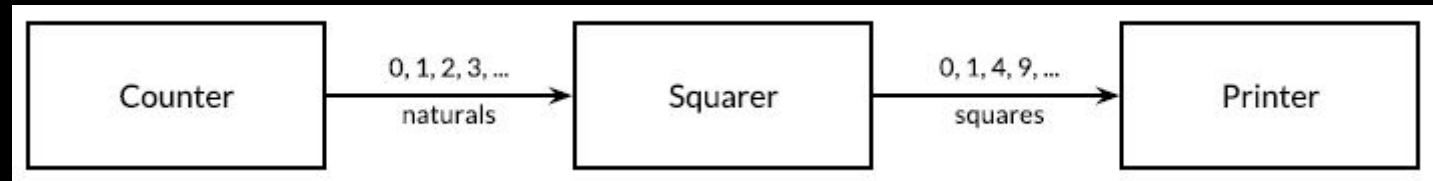
- Range

- Recibe valores repetidamente hasta que eventualmente el channel (`nums`) es cerrado

```
for x := range nums {  
    fmt.Println(i)  
}
```

Concurrencia - Channels

▪ Ejemplo



```
func main() {  
    naturals := make(chan int)  
    squares := make(chan int)  
  
    // Counter  
    go func() {  
        for x := 0; x < 10; x++ {  
            naturals <- x  
        }  
        close(naturals)  
    }()  
}
```

```
// Squarer  
go func() {  
    for x := range naturals {  
        squares <- x * x  
    }  
    close(squares)  
}()  
  
// Printer  
for x := range squares {  
    fmt.Println(x)  
}  
}
```

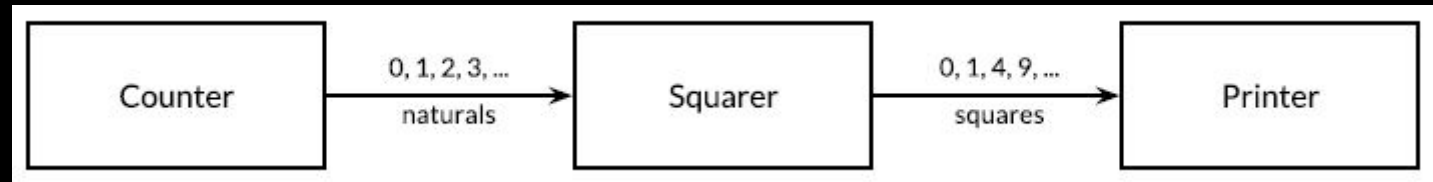
```
0  
1  
4  
9  
16  
25  
36  
49  
64  
81
```


Concurrencia - Channels

- Pueden ser “unidireccionales”
 - Send-only channel
`chan<- int`
 - Receive-only channel
`<-chan int`
- Sólo la goroutine “sender” puede cerrar un send-only channel
- Intentar cerrar un receive-only channel produce en error en tiempo de compilación

Concurrencia - Channels

▪ Ejemplo



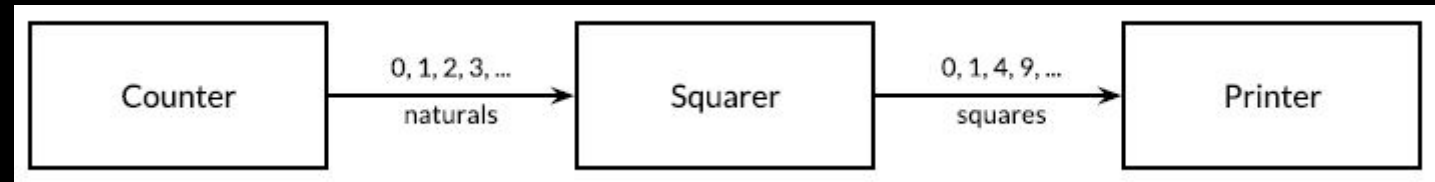
```
func main() {  
    naturals := make(chan int)  
    squares := make(chan int)  
  
    // Counter  
    go func(out chan<- int) {  
        for x := 0; x < 10; x++ {  
            out <- x  
        }  
        close(out)  
    }(naturals)
```

```
    // Squarer  
    go func(in <-chan int, out chan<- int) {  
        for x := range in {  
            out <- x * x  
        }  
        close(out)  
    }(naturals, squares)  
  
    // Printer  
    for x := range squares {  
        fmt.Println(x)  
    }  
}
```

```
0  
1  
4  
9  
16  
25  
36  
49  
64  
81
```

Concurrencia - Channels

▪ Ejemplo



```
func counter(out chan<- int) {
    for x := 0; x < 10; x++ {
        out <- x
    }
    close(out)
}

func squarer(in <-chan int, out chan<- int) {
    for x := range in {
        out <- x * x
    }
    close(out)
}
```

```
func printer(in <-chan int) {
    for x := range in {
        fmt.Println(x)
    }
}

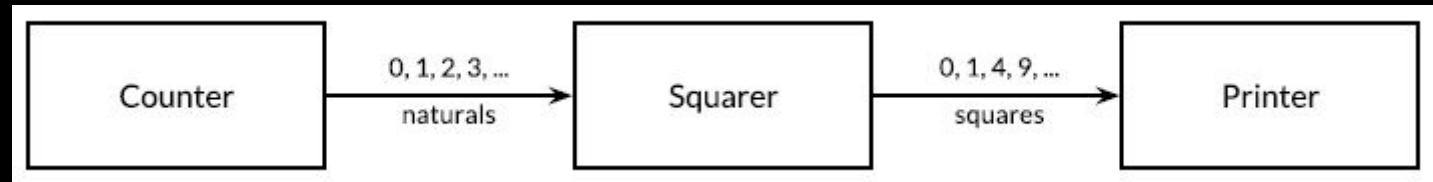
func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go counter(naturals)
    go squarer(naturals, squares)
    printer(squares)
}
```

```
0
1
4
9
16
25
36
49
64
81
```

Concurrencia - Channels

▪ Ejemplo



```
func counter(out chan<- int) {
    for x := 0; x < 10; x++ {
        out <- x
    }
    close(out)
}

func squarer(in <-chan int, out chan<- int) {
    for x := range in {
        out <- x * x
    }
    close(out)
}
```

unidireccional

```
func printer(in <-chan int) {
    for x := range in {
        fmt.Println(x)
    }
}

func main() {
    naturals := make(chan int)
    squares := make(chan int)

    go counter(naturals)
    go squarer(naturals, squares)
    printer(squares)
}
```

unidireccional

bidireccional

0
1
4
9
16
25
36
49
64
81

Concurrencia - Buffered channels

- Buffered channels

```
ch = make(chan string, 3)
```

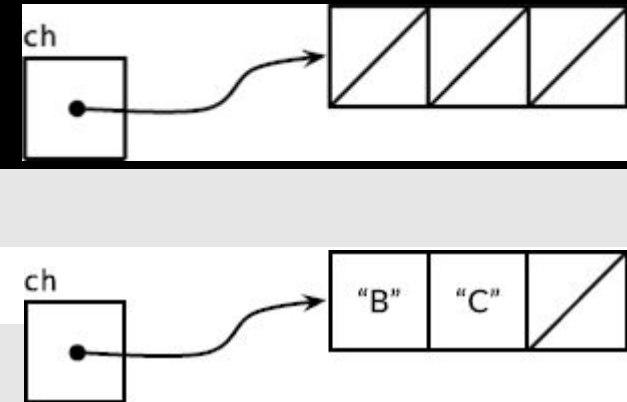
- Tiene asociada una cola de elementos con la capacidad definida en la declaración
- Un “send” agrega un elemento al final de la cola y un “receive” quita y devuelve un elemento del inicio

Concurrencia – Buffered channels

```
ch <- "A"  
ch <- "B"  
ch <- "C"
```

```
fmt.Println(<-ch) // "A"
```

```
fmt.Println(len(ch)) // "2"
```



Concurrencia - Buffered channels

- Productor / Consumidor
 - Un productor genera datos que pone en un buffer
 - Un consumidor saca datos del buffer y los consume

```
func Producer(out chan<- int) {  
    timeProducer := rand.Intn(250)  
    totalProduce := 10  
    for i := 0; i < totalProduce; i++ {  
        time.Sleep(time.Millisecond *  
  
time.Duration(timeProducer))  
        product := rand.Intn(1000)  
        out <- product  
    }  
}
```

```
func Consumer(in <-chan int) {  
    timeConsumer := rand.Intn(1000)  
    for i := range in {  
        time.Sleep(time.Millisecond *  
  
time.Duration(timeConsumer))  
    }  
}
```

prod_cons.go

```
func main() {  
    ch := make(chan int,  
5)  
  
    var wgC sync.WaitGroup  
    wgC.Add(1)  
  
    go func() {  
        Producer(ch)  
        close(ch)  
    }()  
  
    go func() {  
        Consumer(ch)  
        wgC.Done()  
    }()  
  
    wgC.Wait()  
}
```

Concurrencia - Buffered channels

▪ Productores / Consumidores

```
func Producer(out chan<- int) {  
    timeProducer := rand.Intn(250)  
    totalProduce := 10  
    for i := 0; i < totalProduce; i++ {  
        time.Sleep(time.Millisecond *  
  
time.Duration(timeProducer))  
        product := rand.Intn(1000)  
        out <- product  
    }  
}
```

```
func Consumer(in <-chan int) {  
    timeConsumer := rand.Intn(1000)  
    for i := range in {  
        time.Sleep(time.Millisecond *  
  
time.Duration(timeConsumer))  
    }  
}
```

```
func main() {  
    ch := make(chan int)  
    cProd := 2  
    cCons := 5  
  
    var wgP, wgC  
    sync.WaitGroup  
  
    wgP.Add(cProd)  
    wgC.Add(cCons)
```

prod_cons_1.go

```
for c := 1; c <= cCons; c++  
  
    go func(id int) {  
        Consumer(id, ch)  
        wgC.Done()  
    }(c)  
}  
  
for p := 1; p <= cProd; p++  
{  
    go func(id int) {  
        Producer(id, ch)  
        wgP.Done()  
    }(p)  
}  
  
wgP.Wait()  
close(ch)  
wgC.Wait()
```


Concurrencia - Buffered channels

▪ Mirrored request

```
func mirroredQuery() string {
    responses := make(chan string, 3)

    go func() {
        responses <- request("asia.google.com")
    }()

    go func() {
        responses <- request("europe.google.com")
    }()

    go func() {
        responses <- request("americas.google.com")
    }()

    return <-responses // return the quickest
    response
}

func request(hostname string) (response string)
{ /* ... */ }
```

Qué pasaría con un unbuffered channel?

```
func mirroredQuery() string {
    responses := make(chan string)

    go func() {
        responses <- request("asia.google.com")
    }()

    go func() {
        responses <- request("europe.google.com")
    }()

    go func() {
        responses <-
request("americas.google.com")
    }()

    return <-responses
}
```

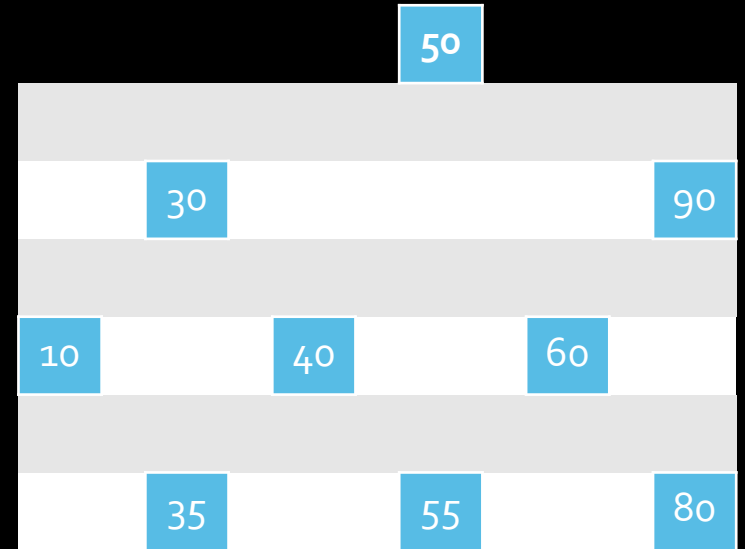
Concurrencia – Concurrencia recursiva

```
func lt(x, y int) bool {
    return x <= y
}

func main() {
    var t *tree.Tree[int]
    for _, i := range []int{50, 30, 90, 40, 60, 10, 80, 35, 55}
    {
        t = t.Insert(i, lt)
    }
    fmt.Println("Tree:", t.GetAll())

    allPaths := t.AllPaths()
    for _, path := range allPaths {
        fmt.Println(path)
    }
}
```

```
Tree: [10 30 35 40 50 55 60 80
90]
Paths:
[50 90]
[50 90 60 80]
[50 90 60 55]
[50 30 40]
[50 30 40 35]
[50 30 10]
```



Concurrencia – Concurrencia recursiva

```
func (t *Tree[T]) AllPaths() [][]T
{
    var paths [][]T
    ch := make(chan []T)

    var wg sync.WaitGroup
    wg.Add(2)

    go func(in <-chan []T) {
        for path := range in {
            paths = append(paths, path)
        }
        wg.Done()
    }(ch)

    go func(ch chan []T) {
        t.finder([]T{}, ch)
        close(ch)
        wg.Done()
    }(ch)

    wg.Wait()
    return paths
}
```

```
func (t *Tree[T]) finder(path []T, out chan<- []T) {
    if t == nil { return }

    path = append(path, t.val)

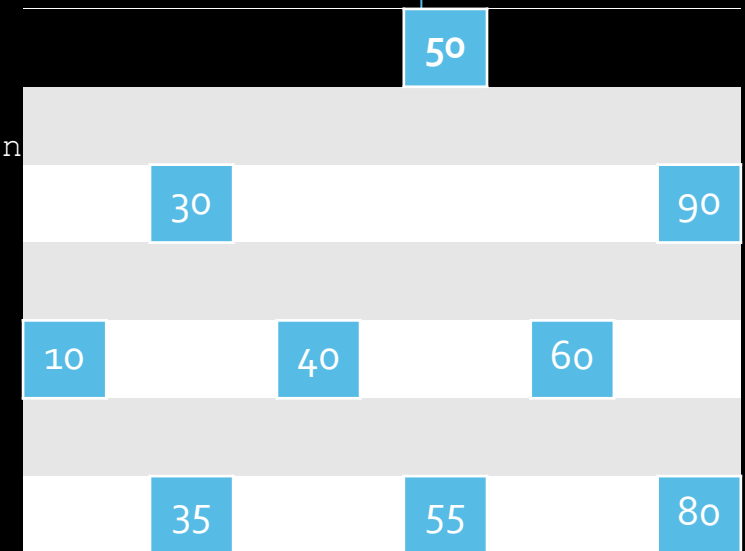
    if t.left == nil || t.right == nil {
        out <- path
    }

    var wgf sync.WaitGroup

    if t.left != nil {
        wgf.Add(1)
        go func() {
            t.left.finder(path, out)
            wgf.Done()
        }()
    }

    if t.right != nil {
        wgf.Add(1)
        go func() {
            t.right.finder(path, out)
            wgf.Done()
        }()
    }

    wgf.Wait()
}
```



Concurrencia - Select

- Select permite que una goroutine espere por más de un channels
 - Send o receive

```
ch1 := make(chan int)
ch2 := make(chan int)

go func() {
    for i := 1; i <= 10; i++
    {
        ch1 <- i
    }
    close(ch1)
}()

go func() {
    for i := 1; i <= 10; i++
    {
        ch2 <- i
    }
    close(ch2)
}()
```

```
var val int
ok1 := true
ok2 := true
for ok1 && ok2 {
    select {
        case val, ok1 =
        <-ch1:
            if ok1 {
                prnt("ch1", val)
            }
        case val, ok2 =
        <-ch2:
            if ok2 {
                prnt("ch2", val)
            }
    }
}
```

```
func prnt(ch string, val int) {
    fmt.Printf("Received from %v: %v", ch,
    val)
}
```

```
if !ok2 {
    for val = range ch1
    {
        prnt("ch1", val)
    }
}
if !ok1 {
    for val = range ch2
    {
        prnt("ch2", val)
    }
}
```

select2.go

Concurrencia - Select

- Problema de los fumadores
 - 3 fumadores alrededor de una mesa
 - Para fumar un cigarrillo se precisa tabaco, papel y fósforo
 - Cada fumador tiene una cantidad ilimitada de un ingrediente
 - Sandy tiene papeles
 - Apple tiene tabaco
 - Daisy tiene fósforos
 - Un “dealer” tiene cantidades ilimitadas de todos los ingredientes
 - El dealer elige al azar un fumador y pone sobre la mesa los dos ingredientes que a dicho fumador le falta
 - El fumador elegido toma los elementos de la mesa, arma su cigarrillo y lo fuma

Concurrencia - Select

▪ Problema de los fumadores

```
const (
    paper = iota
    grass
    match
)

var smokers =
map[int]string{
    paper: "Sandy",
    grass: "Apple",
    match: "Daisy",
}

var wg sync.WaitGroup
```

```
func main() {
    var ingredients [3]chan int
    var signals [3]chan int

    wg.Add(4)
    for i := range smokers {
        ingredients[i] = make(chan int)
        signals[i] = make(chan int)
    }

    for i := range smokers {
        go smoker(i, signals,
ingredients)
    }
    go arbitrate(signals, ingredients)
    wg.Wait()
}
```

smokers.go

Concurrencia - Select

▪ Problema de los fumadores

```
func arbitrate(signals, ingredients [3]chan int)
{
    for i := 0; i < 10; i++ {
        time.Sleep(time.Millisecond * 500)
        next := rand.Intn(3)
        fmt.Println("\nNext:", smokers[next])
        signals[next] <- next
        for c := range ingredients {
            if c != next {
                ingredients[c] <- 1
            }
        }
    }
    for c := range signals {
        close(signals[c])
    }
    wg.Done()
}
```

```
func smoker(id int, signals, ingredients [3]chan int)
{
    count := 0
    for range signals[id] {
        select {
            case <-ingredients[paper]:
            case <-ingredients[grass]:
            case <-ingredients[match]:
        }
        time.Sleep(10 * time.Millisecond)
        select {
            case <-ingredients[paper]:
            case <-ingredients[grass]:
            case <-ingredients[match]:
        }
        time.Sleep(time.Millisecond * 500)
        count++
        fmt.Printf("%v%s smokes %v cigarettes\n",
            strings.Repeat("\t", 3+6*id),
            smokers[id],
            count)
    }
    wg.Done()
}
```

Concurrencia - Select

■ Problema de los fumadores

```
for i := 0; i < 2; i++ {  
    select {  
        case <-ingredients[paper]:  
        case <-ingredients[grass]:  
        case <-ingredients[match]:  
    }  
    time.Sleep(10 *  
time.Millisecond)  
}
```

```
for range signals[id] {  
    for i := range ingredients  
    {  
        if i != id {  
            <-ingredients[i]  
        }  
    }  
}
```

```
func smoker(id int, signals, ingredients [3]chan int)  
{  
    count := 0  
    for range signals[id] {  
        select {  
            case <-ingredients[paper]:  
            case <-ingredients[grass]:  
            case <-ingredients[match]:  
        }  
        select {  
            case <-ingredients[paper]:  
            case <-ingredients[grass]:  
            case <-ingredients[match]:  
        }  
        time.Sleep(time.Millisecond * 500)  
        count++  
        fmt.Printf("%v%s smokes %v cigarettes\n",  
            strings.Repeat("\t", 3+6*id),  
            smokers[id],  
            count)  
    }  
    wg.Done()  
}
```


Concurrencia - Select

▪ Problema de los fumadores

```
func arbitrate(signals, ingredients [3]chan int)
{
    for i := 0; i < 10; i++ {
        time.Sleep(time.Millisecond * 500)
        next := rand.Intn(3)
        fmt.Println("\nNext:", smokers[next])
        signals[next] <- next
        for c := range ingredients {
            if c != next {
                ingredients[c] <- 1
            }
        }
    }
    for c := range signals {
        close(signals[c])
    }
    wg.Done()
}
```

```
select {
case ingredients[paper] <-
1:
case ingredients[grass] <-
1:
case ingredients[match] <-
1:
}
```

```
select {
case ingredients[paper] <-
1:
case ingredients[grass] <-
1:
case ingredients[match] <-
1:
}
```

Concurrencia – Select condicional

- Select puede utilizar una alternativa "default" para send o receive sin bloqueo.

```
ch1 := make(chan int)
ch2 := make(chan int)

go func() {
    for i := 0; i < 10; i++
    {
        ch1 <- 1
    }
    ch1 <- 0
}()

go func() {
    for i := 0; i < 10; i++
    {
        ch2 <- 2
    }
    ch2 <- 0
}()
```

```
fin := 0
for fin < 2 {
    select {
        case val :=
<-ch1:
        if val == 0 {
            fin++
        }
        case val :=
<-ch2:
        if val == 0 {
            fin++
        }
        default:
            // do something
    }
}
```

select3.go

Concurrencia – Select condicional

```
ch1 := make(chan
int)
ch2 := make(chan
int)
var wg
sync.WaitGroup
wg.Add(2)

go func() {
    var val int
    for val != 100 {
        val := <-ch1
    }
    wg.Done()
}()

go func() {
    var val int
    for val != 100 {
        val := <-ch2
    }
    wg.Done()
}()
```

```
for i := 0; i < 20; i++
{
    select {
    case ch1 <- i:
    case ch2 <- i:
    default:
        // do something
    }
    ch1 <- 0
    ch2 <- 0
}
```

select4.go

```
Received from ch2: 2
Received from ch1: 1
Received from ch1: 4
Received from ch2: 6
Received from ch1: 8
Received from ch2: 10
Received from ch1: 12
Received from ch2: 14
Received from ch1: 15
Received from ch2: 19
Received from ch1: 18
Received from ch1: 0
Received from ch2: 0
```

Concurrencia – Exclusión mutua

- Problema de la exclusión mutua

```
var balance int

func Deposit(amount int) {
    balance = balance +
amount
}

func Balance() int {
    return balance
}

func main() {
    for i := 0; i < 10; i++ {
        Deposit(100)
    }
    fmt.Println(Balance())
}
```

1000

```
var balance int

func Deposit(amount int) {
    balance = balance +
amount
}

func Balance() int {
    return balance
}

func main() {
    for i := 0; i < 10; i++ {
        go Deposit(100)
    }
    fmt.Println(Balance())
}
```

900
600
800

bank.go

Concurrencia – Exclusión mutua

▪ Problema de la exclusión mutua - Monitores

```
package bankMonitor

import "fmt"

var deposits = make(chan
int)
var balances = make(chan
int)

func Deposit(amount int) {
    deposits <- amount
}

func Balance() int {
    return <-balances
}
```

```
func teller() {
    var balance int
    for {
        select {
            case amount := <-deposits:
                balance += amount
                fmt.Println("balance:",
balance)
            case balances <- balance:
                }
        }
    }

    func init() {
        go teller()
    }
}
```

bankMonitor.go

```
package main

import (
    "fmt"
    bm "mutex/bankMonitor"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++
    {
        go func() {
            bm.Deposit(100)
            wg.Done()
        }()
    }
    wg.Wait()

    fmt.Println(bm.Balance())
}
```

bank2.go

Concurrencia – Exclusión mutua

- Semáforo binario
- `Type Mutex`
- **Methods:**
 - `func (m *Mutex) Lock()`
 - Bloquea `m`
 - Si `m` ya está bloqueado, la goroutine que invoca a `Lock` se bloquea hasta que otra goroutine invoque a `Unlock`
 - `func (*Mutex) Unlock`
 - Desbloquea `m`
 - Si `m` no está bloqueado se produce un error en tiempo de ejecución

Concurrencia – Exclusión mutua

▪ Semáforo binario

```
import "sync"

var (
    mu sync.Mutex
    balance int
)
```

Variables resguardadas

```
func Deposit(amount int) {
    mu.Lock()
    balance = balance +
amount
    mu.Unlock()
}
```

Sección crítica

```
func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
    return b
}
```

Sección crítica

bankSem.go

```
mu.Lock()
defer
mu.Unlock()
```

return balance

Seminario de Lenguajes opción Go

```
package main

import (
    "fmt"
    bs "mutex/bankSem"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(10)
    for i := 0; i < 10; i++
    {
        go func() {
            bs.Deposit(100)
            wg.Done()
        }()
    }
    wg.Wait()

    fmt.Println(bs.Balance())
}
```

bank3.go

Raúl Champredonde

Concurrencia – Exclusión mutua

- Semáforo binario

```
func Withdraw(amount int) bool {  
    Deposit(-amount)  
    if Balance() < 0 {  
        Deposit(amount)  
        return false // insufficient  
funds  
    }  
    return true  
}
```

No atómico

Concurrencia – Exclusión mutua

▪ Semáforo binario

```
func Withdraw(amount int) bool {  
    Deposit(-amount)  
    if Balance() < 0 {  
        Deposit(amount)  
        return false // insufficient  
funds  
    }  
    return true  
}
```

No atómico

Deadlock

```
var (  
    mu sync.Mutex  
    balance int  
)  
...  
func Deposit(amount int) {  
    mu.Lock()  
    balance = balance + amount  
    mu.Unlock()  
}  
  
func Withdraw(amount int) bool  
{  
    mu.Lock()  
    defer mu.Unlock()  
    Deposit(-amount)  
    if Balance() < 0 {  
        Deposit(amount)  
        return false  
    }  
    return true  
}
```

Concurrencia – Exclusión mutua

▪ Semáforo binario

```
var (  
    mu      sync.Mutex  
    balance int  
)  
  
func Deposit(amount int)  
{  
    mu.Lock()  
    defer mu.Unlock()  
    deposit(amount)  
}  
  
func deposit(amount int)  
{  
    balance += amount  
}
```

bankSem2.go

```
func Balance() int {  
    mu.Lock()  
    defer mu.Unlock()  
    return balance  
}  
  
func Withdraw(amount int) bool  
{  
    mu.Lock()  
    defer mu.Unlock()  
    deposit(-amount)  
    if balance < 0 {  
        deposit(amount)  
        return false  
    }  
    return true  
}
```

```
func main() {  
    var wg sync.WaitGroup  
    wg.Add(15)  
    for i := 0; i < 10; i++  
    {  
        go func() {  
            bs.Deposit(100)  
            wg.Done()  
        }()  
    }  
    for i := 0; i < 5; i++ {  
        go func() {  
            bs.Withdraw(100)  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
  
    fmt.Println(bs.Balance())  
}
```

bank4.go

Concurrencia – Exclusión mutua

- Semáforo “un escritor – múltiples lectores”
- Type `RWMutex`
- Methods:
 - `func (rw *RWMutex) Lock()`
 - Bloquea `rw` para escritura
 - Si `rw` ya está bloqueado para lectura o escritura, la goroutine que invoca a `Lock` se bloquea hasta que otra goroutine invoque a `Unlock` o `RUnlock` según corresponda
 - `func (rw *RWMutex) Unlock()`
 - Desbloquea `rw` para escritura
 - Si `rw` no está bloqueado para escritura se produce un error en tiempo de ejecución
 - `func (rw *RWMutex) RLock()`
 - Bloquea `rw` para lectura
 - Si `rw` ya está bloqueado para escritura, la goroutine que invoca a `RLock` se bloquea hasta que otra goroutine invoque a `Unlock`
 - `func (rw *RWMutex) RUnlock()`
 - Desbloquea `rw` para lectura
 - Si `rw` no está bloqueado para lectura se produce un error en tiempo de ejecución

Concurrencia – Exclusión mutua

- Semáforo “un escritor – múltiples lectores”

```
var (  
    mu      sync.Mutex  
    balance int  
)  
  
func Deposit(amount int)  
{  
    mu.Lock()  
    defer mu.Unlock()  
    deposit(amount)  
}  
  
func deposit(amount int)  
{  
    balance += amount  
}
```

bankSem3.go

```
func Balance() int {  
    mu.RLock()  
    defer mu.RUnlock()  
    return balance  
}  
  
func Withdraw(amount int) bool  
{  
    mu.Lock()  
    defer mu.Unlock()  
    deposit(-amount)  
    if balance < 0 {  
        deposit(amount)  
        return false  
    }  
    return true  
}
```