

Programmeertalen: Go

Een imperatieve gedistribueerde programmeertaal

Koen van Elsen

Universiteit van Amsterdam

10 maart 2020

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

- Ontwikkeld door Google
- Verschenen in 2009
- Gedistribueerd, imperatief
- Statisch en sterk getypeerd
- Eerst compileren, dan runnen

Hulpbronnen

- Documentatie: <https://golang.org/doc/>
- A Tour of Go: <https://tour.golang.org/welcome/1>
- Effective Go: https://golang.org/doc/effective_go.html

Paradigma

- Welk paradigma staat deze week centraal?

Paradigma

- Welk paradigma staat deze week centraal?
- Wat is concurrency?

Paradigma

- Welk paradigma staat deze week centraal?
- Wat is concurrency?
- En wat is parallelisme?

Paradigma

- Welk paradigma staat deze week centraal?
- Wat is concurrency?
- En wat is parallelisme?

[https://en.wikipedia.org/wiki/Concurrency_\(computer_science\)](https://en.wikipedia.org/wiki/Concurrency_(computer_science))

Building Blocks

- Alle standaard dingen: variabelen, if/else, for loop, Arrays (en slices), maps, functies, structs, pointers.

Building Blocks

- Alle standaard dingen: variabelen, if/else, for loop, Arrays (en slices), maps, functies, structs, pointers.
- Go-Routines

Building Blocks

- Alle standaard dingen: variabelen, if/else, for loop, Arrays (en slices), maps, functies, structs, pointers.
- Go-Routines
- Channels

Building Blocks

- Alle standaard dingen: variabelen, if/else, for loop, Arrays (en slices), maps, functies, structs, pointers.
- Go-Routines
- Channels
- (B)Locking & Deadlock
- Select

Hello World

```
package main
```

```
import (  
    "fmt"  
)
```

```
func main() {  
    fmt.Println("Hello world!")  
}
```

```
$ go run helloworld.go  
Hello world!
```

Variabelen declareren

- Eén variabele tegelijk:

```
var i int
```

Variabelen declareren

- Eén variabele tegelijk:

```
var i int
```

- Meerdere variabelen van hetzelfde type tegelijkertijd:

```
var foo bar bool
```

Variabelen declareren

- Eén variabele tegelijk:
`var i int`
- Meerdere variabelen van hetzelfde type tegelijkertijd:
`var foo bar bool`
- Met initialisatie én type:
`var i, j int = 1, 2`

Variabelen declareren

- Eén variabele tegelijk:

```
var i int
```

- Meerdere variabelen van hetzelfde type tegelijkertijd:

```
var foo bar bool
```

- Met initialisatie én type:

```
var i, j int = 1, 2
```

- Met initialisatie zonder type:

```
var i, j = 1, 2
```

Variabelen declareren (2): binnen een functie

Binnen een functie kun je ook gebruik maken van *short assignment*

- Met één variabele:

```
i := 0
```

Variabelen declareren (2): binnen een functie

Binnen een functie kun je ook gebruik maken van *short assignment*

- Met één variabele:

```
i := 0
```

- Of meerdere variabelen (mogelijk ook van van verschillende types):

```
i, bericht := 2, "bla"
```

Typen van variabelen

Go heeft diverse primitieve datatypes waaronder:

- bool
- string
- int
- uint
- float32, float64
- complex64, complex128

Typen van variabelen

Go heeft diverse primitieve datatypes waaronder:

- `bool`
- `string`
- `int`
- `uint`
- `float32`, `float64`
- `complex64`, `complex128`

En diverse andere integer-types (zoals `uint16`). In tegenstelling tot C worden variabelen standaard geïnitieerd op een standaardwaarde (zoals `0`, `0.0` of `false`).

Typen van variabelen

Go heeft diverse primitieve datatypes waaronder:

- bool
- string
- int
- uint
- float32, float64
- complex64, complex128

En diverse andere integer-types (zoals uint16). In tegenstelling tot C worden variabelen standaard geïnitieerd op een standaardwaarde (zoals 0, 0.0 of false).

Door de naam van het type te gebruiken als functie kun je numerieke typen heen en weer casten:

```
pi := 3.1415  
fmt.Println(int(pi))
```

3

Functie-definitie en aanroep

```
package main

import "fmt"

func add(x, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(8, 7))
}
```

Meerdere return values

```
func verwissel(a, b int) (int, int) {  
    return b, a  
}
```

```
func main() {  
    fmt.Println(verwissel(3,4))  
}
```

4 3

Standaard return values

```
func verwissel(a, b int) (c, d int) {  
    c = b  
    d = a  
    return  
}
```

```
func main() {  
    fmt.Println(verwissel(3,4))  
}
```

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

For-loop en if-statement

De for-loop is de enige loop in Go (en vervangt de while en foreach-loop).

```
func main() {  
    sum := 0  
    for i := 1; i <= 10; i++ {  
        sum += i  
        if i%2 == 0 {  
            fmt.Println("Tussenstap:", i, "Som:", sum)  
        }  
    }  
}
```

For-loop en if-statement

De for-loop is de enige loop in Go (en vervangt de while en foreach-loop).

```
func main() {  
    sum := 0  
    for i := 1; i <= 10; i++ {  
        sum += i  
        if i%2 == 0 {  
            fmt.Println("Tussenstap:", i, "Som:", sum)  
        }  
    }  
}
```

```
Tussenstap: 2  Som: 3  
Tussenstap: 4  Som: 10  
Tussenstap: 6  Som: 21  
Tussenstap: 8  Som: 36  
Tussenstap: 10 Som: 55
```

For-loop (2)

Je kunt ook de init en post-operaties laten vervallen, zodat het gedrag hetzelfde is als dat van een while-loop in andere talen:

```
func main() {  
    a := 10  
    for a > 0 {  
        a++  
    }  
    fmt.Println(a)  
}
```

For-loop (2)

Je kunt ook de init en post-operaties laten vervallen, zodat het gedrag hetzelfde is als dat van een while-loop in andere talen:

```
func main() {  
    a := 10  
    for a > 0 {  
        a++  
    }  
    fmt.Println(a)  
}
```

—2147483648

For-loop (3)

Zonder inhoud gedraagt een for-loop zich als een while(true):

```
func main() {  
    i := 10  
  
    for {  
        if i % 250 == 0 {  
            break  
        } else {  
            i++  
        }  
    }  
  
    fmt.Println(i)  
}
```

For-loop (3)

Zonder inhoud gedraagt een for-loop zich als een while(true):

```
func main() {  
    i := 10  
  
    for {  
        if i % 250 == 0 {  
            break  
        } else {  
            i++  
        }  
    }  
  
    fmt.Println(i)  
}
```

250

Arrays

- In Go kun je eenvoudig een array declareren:

```
var a [10]int
```

Arrays

- In Go kun je eenvoudig een array declareren:

```
var a [10]int
```

- Je kunt de elementen vervolgens net als in C een waarde geven, meerdere elementen tegelijkertijd:

```
a[0] = 1
```

```
a[9] = 10
```

```
a[2], a[3] = 4, 6
```

Arrays

- In Go kun je eenvoudig een array declareren:

```
var a [10]int
```

- Je kunt de elementen vervolgens net als in C een waarde geven, meerdere elementen tegelijkertijd:

```
a[0] = 1
```

```
a[9] = 10
```

```
a[2], a[3] = 4, 6
```

- Of de grootte van de array opvragen en eroverheen lopen:

```
primes := [6]int{2, 3, 5, 7, 11, 13}
```

```
    for i := 0; i < len(primes); i++ {
```

```
        fmt.Println(primes[i])
```

```
    }
```

Arrays (2)

Een array is een data-type en niet zoals in C een pointer. Als je een array meegeeft aan een functie, of als waarde toekent dan maakt Go een kopie van de array.

Arrays (2)

Een array is een data-type en niet zoals in C een pointer. Als je een array meegeeft aan een functie, of als waarde toekent dan maakt Go een kopie van de array.

```
func aanpassen(primes [6]int) {  
    primes[0] = 42  
}  
  
func main() {  
    primes := [6]int{2, 3, 5, 7, 11, 13}  
    fmt.Println(primes)  
    aanpassen(primes)  
    fmt.Println(primes)  
}
```

Arrays (2)

Een array is een data-type en niet zoals in C een pointer. Als je een array meegeeft aan een functie, of als waarde toekent dan maakt Go een kopie van de array.

```
func aanpassen(primes [6]int) {  
    primes[0] = 42  
}  
  
func main() {  
    primes := [6]int{2, 3, 5, 7, 11, 13}  
    fmt.Println(primes)  
    aanpassen(primes)  
    fmt.Println(primes)  
}
```

```
[2 3 5 7 11 13]  
[2 3 5 7 11 13]
```

Slices

Omdat arrays niet zo heel flexibel zijn, kent Go ook slices. Een slice heeft geen vaste grootte maar representeert een (gedeelte van) een onderliggend array. Een slice heeft een lengte en een (maximale) capaciteit. De capaciteit is afhankelijk van de grootte van de achterliggende array.

Slices

Omdat arrays niet zo heel flexibel zijn, kent Go ook slices. Een slice heeft geen vaste grootte maar representeert een (gedeelte van) een onderliggend array. Een slice heeft een lengte en een (maximale) capaciteit. De capaciteit is afhankelijk van de grootte van de achterliggende array.

```
func aanpassen(primes []int) {  
    primes[0] = 42  
}  
  
func main() {  
    primes := [6]int{2, 3, 5, 7, 11, 13}  
    slice := primes[3:]  
    fmt.Println(primes)  
    aanpassen(slice)  
    fmt.Println(primes)  
}
```


Slices

Omdat arrays niet zo heel flexibel zijn, kent Go ook slices. Een slice heeft geen vaste grootte maar representeert een (gedeelte van) een onderliggend array. Een slice heeft een lengte en een (maximale) capaciteit. De capaciteit is afhankelijk van de grootte van de achterliggende array.

```
func aanpassen(primes []int) {  
    primes[0] = 42  
}  
  
func main() {  
    primes := [6]int{2, 3, 5, 7, 11, 13}  
    slice := primes[3:]  
    fmt.Println(primes)  
    aanpassen(slice)  
    fmt.Println(primes)  
}
```

```
[2 3 5 7 11 13]  
[2 3 5 42 11 13]
```

Slices (2)

```
func main() {  
    s := []int{2, 3, 5, 7, 11, 13}  
    printSlice(s)  
    // Slice the slice to give it zero length.  
    s = s[:0]  
    printSlice(s)  
    // Extend its length.  
    s = s[:4]  
    printSlice(s)  
    // Drop its first two values.  
    s = s[2:]  
    printSlice(s)  
}  
  
func printSlice(s []int) {  
    fmt.Printf("len=%d cap=%d %v\n", len(s), cap(s), s)  
}
```

Slices (3)

Tevens zijn er nog 3 ingebouwde functies voor slices:

- `make([]T, length, capacity)`
- `append(s []T, vs ...T) []T`
 - ▶ Omdat `append` de onderliggende array kan her-alloceren is het belangrijk het resultaat op te slaan
- `copy(dst, src []T) int`

Slices (3)

Tevens zijn er nog 3 ingebouwde functies voor slices:

- `make([]T, length, capacity)`
- `append(s []T, vs ...T) []T`
 - ▶ Omdat `append` de onderliggende array kan her-alloceren is het belangrijk het resultaat op te slaan
- `copy(dst, src []T) int`

```
var s []int
s = make([]int, 3, 6) /* A slice of length 3 backed by an anonymous array of
                       length 6 (the capacity of the slice). */

t := make([]int, 2)   // Shorthand, leaving out the capacity
t = append(t, 42, 54) // Automatic reallocation to a new array
copy(s, t)            // copy returns 3; s becomes {0, 0, 42}
u := append(s, 91)    // The same array as for 's', but a different length
s = s[len(s):cap(u)]  // s becomes {91, 0, 0}
```

Slices (3)

Tevens zijn er nog 3 ingebouwde functies voor slices:

- `make([]T, length, capacity)`
- `append(s []T, vs ...T) []T`
 - ▶ Omdat `append` de onderliggende array kan her-alloceren is het belangrijk het resultaat op te slaan
- `copy(dst, src []T) int`

```
var s []int
s = make([]int, 3, 6) /* A slice of length 3 backed by an anonymous array of
                       length 6 (the capacity of the slice). */

t := make([]int, 2)  // Shorthand, leaving out the capacity
t = append(t, 42, 54) // Automatic reallocation to a new array
copy(s, t)           // copy returns 3; s becomes {0, 0, 42}
u := append(s, 91)    // The same array as for 's', but a different length
s = s[len(s):cap(u)]  // s becomes {91, 0, 0}
```

Lees hier meer over slices: <https://blog.golang.org/go-slices-usage-and-internals>

For-loop over slices met range

Met behulp van `range` kun je over een slice heen lopen, je krijgt dan iedere iteratie twee waarden terug: de index en een kopie van het desbetreffende element uit de slice.

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}
```

```
func main() {  
    for i, v := range pow {  
        fmt.Printf("2**%d = %d\n", i, v)  
    }  
}
```

For-loop over slices met range

Met behulp van `range` kun je over een slice heen lopen, je krijgt dan iedere iteratie twee waarden terug: de index en een kopie van het desbetreffende element uit de slice.

```
var pow = []int{1, 2, 4, 8, 16, 32, 64, 128}
```

```
func main() {  
    for i, v := range pow {  
        fmt.Printf("2**%d = %d\n", i, v)  
    }  
}
```

```
2**0 = 1  
2**1 = 2  
2**2 = 4  
2**3 = 8  
..
```

Errors

- Go kent geen try catch blokken zoals bijvoorbeeld Python of Java.

Errors

- Go kent geen try catch blokken zoals bijvoorbeeld Python of Java.
- Functies waarbinnen iets fout kan gaan geven naast de return waarde ook een type error terug.

```
file, err := os.Open("file.go") // For read access.  
if err != nil {  
    log.Fatal(err)  
}
```

Errors

- Go kent geen try catch blokken zoals bijvoorbeeld Python of Java.
- Functies waarbinnen iets fout kan gaan geven naast de return waarde ook een type error terug.

```
file, err := os.Open("file.go") // For read access.  
if err != nil {  
    log.Fatal(err)  
}
```

- Je kan zelf ook Error types implementeren zolang ze voldoen aan de error interface.

```
type error interface {  
    Error() string  
}
```

Errors (2)

- Om de control flow makkelijker te maken kent go het `defer` statement. Deze statements worden uitgevoerd na de `return`.

Errors (2)

- Om de control flow makkelijker te maken kent go het `defer` statement. Deze statements worden uitgevoerd na de `return`.
- Runtime errors worden een `Panic` genoemd.

Errors (2)

- Om de control flow makkelijker te maken kent go het `defer` statement. Deze statements worden uitgevoerd na de `return`.
- Runtime errors worden een `Panic` genoemd.
- Als je een panic niet Recoverd crashed het programma.

Errors (2)

- Om de control flow makkelijker te maken kent go het defer statement. Deze statements worden uitgevoerd na de return.
- Runtime errors worden een Panic genoemd.
- Als je een panic niet Recoverd crashed het programma.
- Deferred statement worden ook na een panic uitgevoerd.

```
func a() {  
    i := 0  
    defer fmt.Println(i)  
    i++  
    return  
}
```

Errors (2)

- Om de control flow makkelijker te maken kent go het defer statement. Deze statements worden uitgevoerd na de return.
- Runtime errors worden een Panic genoemd.
- Als je een panic niet Recoverd crashed het programma.
- Deferred statement worden ook na een panic uitgevoerd.

```
func a() {  
    i := 0  
    defer fmt.Println(i)  
    i++  
    return  
}
```

Errors (2)

- Om de control flow makkelijker te maken kent go het defer statement. Deze statements worden uitgevoerd na de return.
- Runtime errors worden een Panic genoemd.
- Als je een panic niet Recoverd crashed het programma.
- Deferred statement worden ook na een panic uitgevoerd.

```
func a() {  
    i := 0  
    defer fmt.Println(i)  
    i++  
    return  
}
```

- De argumenten worden gevalueerd tijdens de evaluatie van het defer statement.

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

Structs

Vergelijkbaar met het type in C kun je een struct definiëren:

```
type Position struct {  
    Row, Col int  
}
```

```
func main() {  
    pos1 := Position{3, 4}  
    pos2 := Position{}  
    pos3 := Position{Col: 4}  
    pos1.Col++  
    fmt.Println("Row:", pos1.Row, "Column:", pos1.Col)  
    fmt.Println("Row:", pos2.Row, "Column:", pos2.Col)  
    fmt.Println("Row:", pos3.Row, "Column:", pos3.Col)  
}
```

Structs

Vergelijkbaar met het type in C kun je een struct definiëren:

```
type Position struct {  
    Row, Col int  
}
```

```
func main() {  
    pos1 := Position{3, 4}  
    pos2 := Position{}  
    pos3 := Position{Col: 4}  
    pos1.Col++  
    fmt.Println("Row:", pos1.Row, "Column:", pos1.Col)  
    fmt.Println("Row:", pos2.Row, "Column:", pos2.Col)  
    fmt.Println("Row:", pos3.Row, "Column:", pos3.Col)  
}
```

```
Row: 3 Column: 5  
Row: 0 Column: 0  
Row: 0 Column: 4
```

Structs (2)

En je kunt vervolgens ook functies definiëren die je kunt aanroepen op een instantie van zo'n struct, ofwel een methode:

```
type Position struct {  
    Row, Col int  
}  
  
func (pos Position) Print() {  
    fmt.Println("Row:", pos.Row, "Column:", pos.Col)  
}  
  
func main() {  
    pos := Position{3, 4}  
    pos.Print()  
}
```

Structs (2)

En je kunt vervolgens ook functies definiëren die je kunt aanroepen op een instantie van zo'n struct, ofwel een methode:

```
type Position struct {  
    Row, Col int  
}  
  
func (pos Position) Print() {  
    fmt.Println("Row:", pos.Row, "Column:", pos.Col)  
}  
  
func main() {  
    pos := Position{3, 4}  
    pos.Print()  
}
```

Row: 3 Column: 4

Type aliases

En dit kan ook voor zelf gedefinieerde types:

```
type MyFloat float64
```

```
func (f MyFloat) Abs() float64 {  
    if f < 0 {  
        return float64(-f)  
    }  
    return float64(f)  
}
```

```
func main() {  
    f := MyFloat(-math.Sqrt2)  
    fmt.Println(f.Abs())  
}
```

Zie ook <https://golang.org/pkg/math/> voor meer informatie over math constants.

Funcities als typen (1)

Zoals je een functie-definitie schrijft, kun je dit ook gebruiken als type voor een parameter voor een andere functie:

```
func compute(powerrrr func(float64, float64) float64, x, y float64) {  
    fmt.Println(powerrrr(x, y))  
}  
  
func main() {  
    fmt.Println(math.Pow(2, 3))  
    compute(math.Pow, 2, 3)  
}
```

Funcities als typen (1)

Zoals je een functie-definitie schrijft, kun je dit ook gebruiken als type voor een parameter voor een andere functie:

```
func compute(powerrrr func(float64, float64) float64, x, y float64) {  
    fmt.Println(powerrrr(x, y))  
}
```

```
func main() {  
    fmt.Println(math.Pow(2, 3))  
    compute(math.Pow, 2, 3)  
}
```

```
8  
8
```


Functies als typen (2)

```
func funky(slice []int) func(int) int {  
    closure := func(i int) int {  
        return slice[i] // Access to a variable from the encompassing scope  
    }  
    return closure // Returning a function is not what makes it a closure  
}
```

```
func main() {  
    f := funky([]int{91, 42, 54})  
    g := funky([]int{91, 43, 93})  
    for i := 0; i < 3; i++ {  
        fmt.Println(f(i) == g(i))  
    }  
}
```

Funcities als typen (2)

```
func funky(slice []int) func(int) int {  
    closure := func(i int) int {  
        return slice[i] // Access to a variable from the encompassing scope  
    }  
    return closure // Returning a function is not what makes it a closure  
}
```

```
func main() {  
    f := funky([]int{91, 42, 54})  
    g := funky([]int{91, 43, 93})  
    for i := 0; i < 3; i++ {  
        fmt.Println(f(i) == g(i))  
    }  
}
```

```
true  
false  
false
```

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

Go-routines

```
func say(s string) {  
    for i := 0; i < 5; i++ {  
        time.Sleep(100 * time.Millisecond)  
        fmt.Println(s)  
    }  
}  
  
func main() {  
    go say("world")  
    say("hello")  
}
```

Wat verwacht je dat hier uit komt?

Shared memory en data races

```
var x int = 2

func double() {
    y := x
    // 'x' and 'y' are equal, right?
    x += y
}

func main() {
    go double()
    go double()
    time.Sleep(time.Second) // Never wait for your goroutines like this
    fmt.Println(x)
}
```

Channels

Om te communiceren tussen goroutines kun je gebruik maken van **channels**:

```
var v int
var c chan int    // Channels can be defined for every type, even channels
c = make(chan int) // Channels are created using make
c <- v            // Send 'v' to the channel
v = <-c           // Receive from the channel and store in 'v'
```

Channels (2)

```
func count(c chan int) {  
    n := 0  
    for { // An infinite loop  
        c <- n  
        n++  
    }  
}  
  
func main() {  
    var c chan int  
    c = make(chan int)  
    go count(c) // Mind the goroutine: count does not return  
    fmt.Println(<-c, <-c, <-c)  
}
```

Channels (2)

```
func count(c chan int) {  
    n := 0  
    for { // An infinite loop  
        c <- n  
        n++  
    }  
}  
  
func main() {  
    var c chan int  
    c = make(chan int)  
    go count(c) // Mind the goroutine: count does not return  
    fmt.Println(<-c, <-c, <-c)  
}
```

0 1 2

Channels (3)

Het schrijven naar, en lezen van een channel is **blocking**.

```
c := make(chan int)
c <- 6
fmt.Println(<-c) // Unreachable
```

Tenzij je channel een buffer heeft, dan is het alleen blocking als het channel volzit:

```
c := make(chan int, 1)
c <- 6
fmt.Println(<-c)
```

Wachten op go-routines

```
var done chan int = make(chan int)

func hello() {
    fmt.Println("Hello World!")
    done <- 1 // The 1 carries no meaning
}

func main() {
    // Spawn two goroutines
    go hello()
    go hello()
    // Wait for them to finish
    <-done
    <-done
}
```

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

Bomen representeren met een 2d-array

```
0
|-- 1
|  |-- 4
|-- 2
|-- 3
|  |-- 5
|  |-- 6
```

```
tree := [] []int{{1, 2, 3}, // the children of node 0
                {4},         // the children of node 1
                {},           // etc.
                {5, 6},
                {}, {}, {}}
```

Doorzoeken van de boom met go-routines

```
var leaves func(int)
queue := make(chan int)

leaves = func(node int) {
    if len(tree[node]) == 0 {
        fmt.Println(node, "is a leaf")
    } else {
        for _, child := range tree[node] {
            go leaves(child)
        }
    }
    queue <- node
}

go leaves(0)
for i := 0; i < len(tree); i++ {
    fmt.Println("Processed node", <-queue)
}
```

Doorzoeken van de boom met go-routines (2)

```
leaves = func(node int) {  
    if len(tree[node]) == 0 {  
        fmt.Println(node, "is a leaf")  
    } else {  
        for _, child := range(tree[node]) {  
            go leaves(child)  
        }  
    }  
    queue <- node  
}
```

```
Processed node 0  
Processed node 1  
2 is a leaf  
Processed node 2  
Processed node 3  
4 is a leaf  
Processed node 4  
5 is a leaf  
Processed node 5  
6 is a leaf  
Processed node 6
```

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

Getal raden

- Gedistribueerd raden van een getal onder de 100
- Routines raden een willekeurig aantal getallen
- Voor elke foute poging wordt een nieuwe raad-routine opgestart
- Na een correcte poging, worden geen raad-routines meer opgestart

Hoe wachten we tot alle routines zijn afgelopen?

Radende functie

```
import "math/rand"

var c chan int = make(chan int, 100)
var done chan int = make(chan int)

func guess() {
    for i := rand.Intn(3) + 1; i > 0; i-- { // At most 3 guesses
        c <- rand.Intn(100)
    }
    done <- 1
}
```

```

rand.Seed(time.Now().UTC().UnixNano())
var runningCount int
secret := rand.Intn(100)
guessed := false
runningCount += 1
go guess()
for runningCount != 0 {
    select { // Execute a case that is ready, pick a random one if multiple are
        case g := <-c:
            if guessed == false { // Ignore all guesses after a correct guess
                if g == secret {
                    guessed = true
                } else {
                    runningCount++
                    go guess()
                }
            }
        case <-done:
            runningCount--
    }
}
fmt.Println("Guessed: ", guessed)

```

Het werkt niet!

Waarom?

- De Guess-channel had een buffer
- De Done-channel werd geleegd
 - ▶ Er bleven geen guess-routines meer draaien
 - ▶ In de buffer van het guess-channel zaten nog steeds getallen

Het werkt niet!

Waarom?

- De Guess-channel had een buffer
- De Done-channel werd geleegd
 - ▶ Er bleven geen guess-routines meer draaien
 - ▶ In de buffer van het guess-channel zaten nog steeds getallen

Oplossing: raad-routines in leven houden totdat er nieuwe zijn gemaakt.

Lees meer hier: <https://blog.golang.org/pipelines>

Poging 2

```
import "math/rand"

type Guess struct {
    Value int
    Ack chan int
}

var c chan Guess = make(chan Guess, 100)
var done chan int = make(chan int)

func guess() {
    ack := make(chan int)
    for i := rand.Intn(3) + 1; i > 0; i-- { // At most 3 guesses
        c <- Guess{rand.Intn(100), ack}
        <-ack // Wait until the guess is processed
    }
    done <- 1
}
```

```

var runningCount int
secret := rand.Intn(100)
guessed := false

runningCount += 1
go guess()
for runningCount != 0 {
    select { // Execute a case that is ready, pick a random one if multiple are
        case g := <-c:
            if guessed == false { // Ignore all guesses after a correct guess
                if g.Value == secret {
                    guessed = true
                } else {
                    runningCount++
                    go guess()
                }
            }
            g.Ack <- 1
        case <-done:
            runningCount--
    }
}

```

Inhoud

Introductie

If-statements, arrays, slices en loops

Fouten afhandelen

Structs, Type aliases en functies als typen

Go-routines en channels

Zoekbomen met Go

Voorbeeld: getallen raden

Mazes als zoekbomen

Doolhof als zoekboom

```
+---+---+---+
| A | B   C
+   +---+---+
| D   E   F
+   +---+   +
| G   H   I |
+---+---+   +
```

```
A
`-- D
    |-- E
    |   `-- F
    |       |-- :-)
    |       `-- I
    |           `-- :-)
    `-- G
        `-- H
```


sync.Once: eenmalige initialisatie

```
import "sync" // https://golang.org/pkg/sync/
import "fmt"

var s []int
var initialise sync.Once

func fill() {
    fmt.Println("Initialising!")
    s = []int{91, 42, 54}
}

func getIndex(x int, result chan int) {
    initialise.Do(fill) // fill is only executed at the first getIndex invocation
    for i, v := range s {
        if x == v {
            result <- i
            return
        }
    }
    result <- -1 // Not found
}
```

Volgende week, laatste week :-)

C++!