Université Paris Cité

Cours Go

Jean-Baptiste Yunès

yunes.informatique.univ-paris-diderot.fr

Jean-Baptiste.Yunes@u-pariscite.fr

v26.09.09

Informations

- Ce cours n'est pas un cours complet/exhaustif de Go
- Certaines constructions seront à acquérir par vous-même
- Consultez le site `https://go.dev/`
  - Installez la version la plus récente (aujourd'hui 1.27.1)
- Installez un IDE
  - Visual Studio Code est gratuit et open source
  - Installez des extensions pour Go (au moins Go)
- Ce support contient des éléments extraits de la documentation Go

Anatomie d'un projet Go

Généralités

- Module
  - Ensemble de packages
  - définit par un fichier `go.mod`
- Package
  - Ensemble de codes sources
- Code source
  - directive `package` en tête

Anatomie d'un projet Go

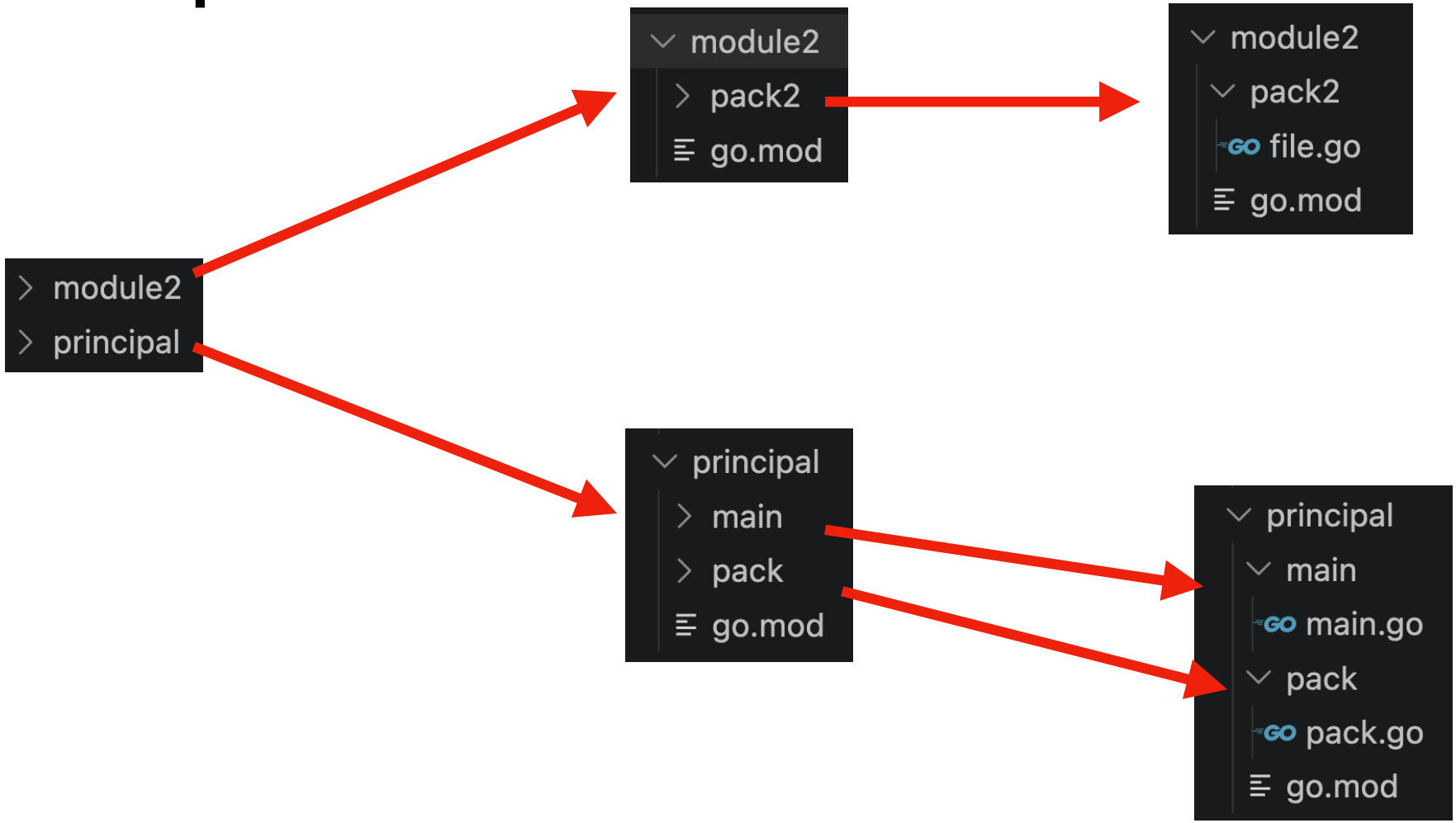
Généralités

- Un module <=> un répertoire + `go.mod`
- Un package <=> un répertoire à l'intérieur d'un module
- Un ou plusieurs fichiers sources dans un package avec directive `package`

- Il vaut mieux que les modules et packages soient dans des répertoires qui portent leur nom

Anatomie d'un projet Go

Exemple



Anatomie d'un projet Go

`go.mod`

- Peut être créé à la main ou
- par la commande : `go mod init` *`nom_du_module`*
- Un exemple de fichier `go.mod` :

```
module module2

go 1.27.0
```

Anatomie d'un projet Go

Un code source

- Un exemple de fichier `.go` :

```go
package main

import (
    "fmt"
    "main/pack"
    "module2/pack2"
)

func main() {
    fmt.Println("cuicui")
    pack.F()
    pack2.F()
}
```

Anatomie d'un projet Go

`go.work`

- Il est parfois nécessaire d'utiliser un fichier `go.work`
- On peut le créer à la main ou
- utiliser la commande : `go work init`
- Un exemple :
  - `use` permet de faire en sorte que la commande go retrouve ses petits

```
go 1.27.0

use principal

use module2
```

Anatomie d’un projet Go

main…

- Un programme Go démarre par l’exécution de la fonction `main` définie dans le package `main`

Exercice

- Créer quelque part un exemple minimal :
  - un fichier `test.go` contenant la fonction `main` du package `main` qui à l'exécution affichera `Bonjour`
  - Une exécution est pbtenue par la commande `go run` *`fichier.go`*
- Attention, il ne s'agit pas encore de faire un projet modulaire mais de tester si l'installation fonctionne
  - Dans la suite on veillera à toujours utiliser les modules…

La commande Go

- La commande Go est versatile, on l'utilise avec des sous-commandes
  - `go run`
  - `go build`
  - `go mod`
  - `go work`
  - `go clean`
  - `go test`
- Pour l'aide : `go help` *`sous-commande`*

Le langage Go

Quelques élements syntaxiques

```go
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    var val1, val2 float32
    val1, val2 = initialValue()
    affiche(val1, val2)
}

func initialValue() (float32, float32) {
    v1 := rand.Float32()
    v2 := rand.Float32()
    return v1, v2
}

func affiche(v1, v2 float32) {
    fmt.Println(v1, " -- ", v2)
}
```

Le langage Go

Quelques élements syntaxiques

- Un code **doit** être placé dans un package
- Si on utilise d'autres packages il faut les importer
  - On ne peut importer des packages que l'on utilise pas
    - Erreur de compilation
- Ensuite on définit des types, des fonctions, des variables globales

Le langage Go

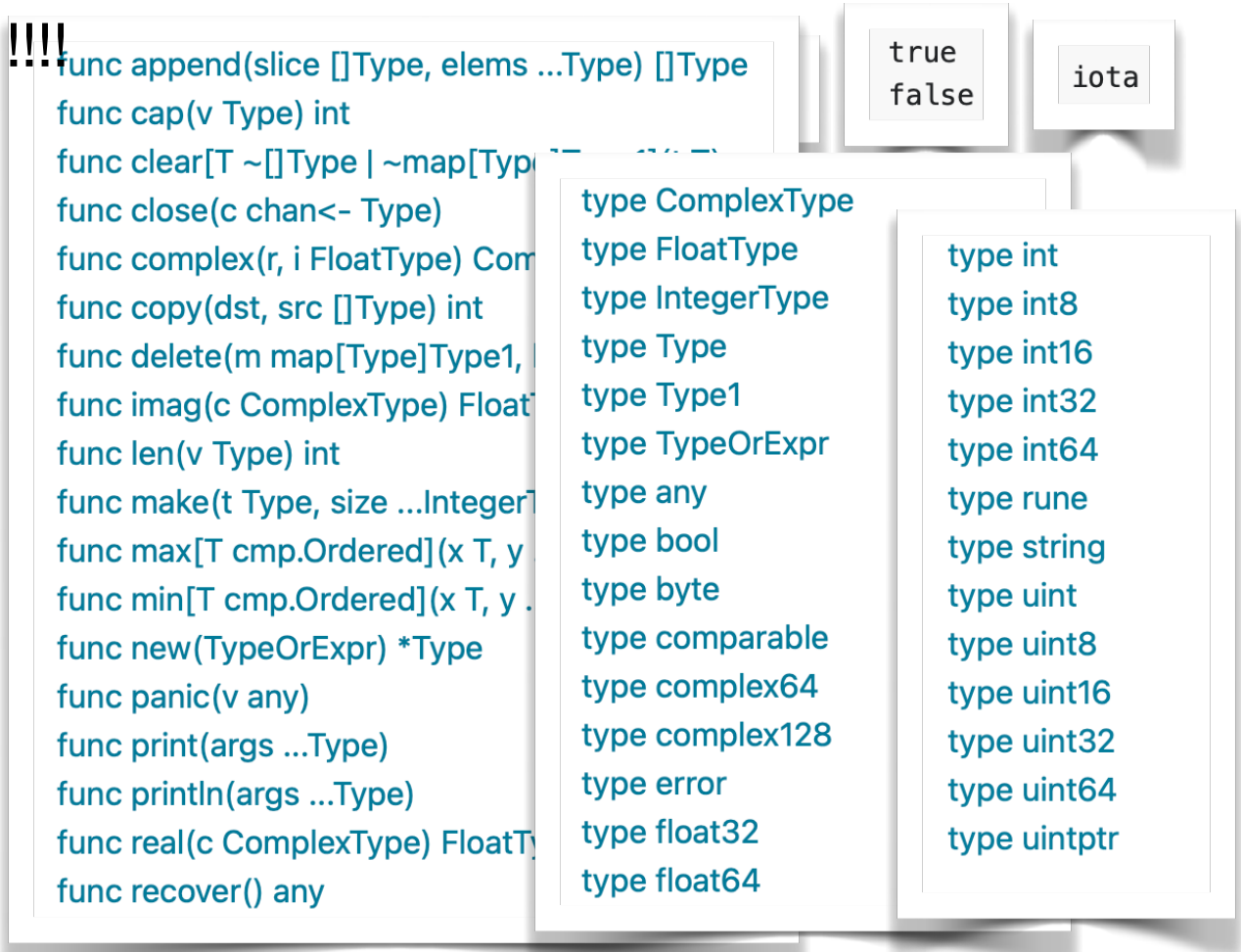
Règle de visibilité

- Le découpage en package permet de rendre visible/cacher des éléments
- Un élément d'un package n'est **visible** depuis un autre package que si le nom de cet élément commence par une **majuscule**
- On ne peut utiliser un élément définit dans un autre package que si l'on importe le package correspondant (et si le nom de l'élément commence par une majuscule)
- Attention cette règle ne fonctionne pas pour le «package» `builtin` qui contient des éléments de base

Le langage Go

builtin

- Les mots prédéclarés de Go
- Ils ne sont pas réservés !!!!
  - On peut les redéfinir



Le langage Go

Les mots-clés de Go

- Ces mots-clés ne peuvent être employés pour autre chose que ce pourquoi ils sont définis

break	case	chan	const	continue	default
defer	else	fallthrough	for	func	go
goto	if	import	interface	map	package
range	return	select	struct	switch	type
var					

Le langage Go

Les identificateurs

- Un identificateur (de fonction, variable, type, etc) peut contenir n’importe quel caractère alphabétique et numérique Unicode ainsi que le caractère _
    - එඑෙමᚢᚤ23 est valide (Tamoul+Runique/Chiffres) mais déconseillé
    - si la première lettre est une majuscule (règle de visibilité)
- Convention camel case : ilFaitBeauEtChaud
    - les acronymes sont en majuscules : sendJSON

Le langage Go

Les opérateurs

```
+    &     +=    &=     &&    ==    !=    (    )
-    |     -=    |=     ||    <     <=    [    ]
*    ^     *=    ^=     <-    >     >=    {    }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     ...   .    :
     &^          &^=          ~
```

Le langage Go

Les litéraux

- Les littéraux:
  - entier : 666
  - flottant : 3.14
  - complexe : 1+3i
  - string : "Bonjour"
  - rune : 'A'

Le langage Go

Le formatage

- Le formatage est libre mais :
  - Go rajoute implicitement un ; :
    - après un identificateur ou un littéral en fin de ligne
    - après `break`, `continue`, `fallthrough`, `return`, `++`, `--`, `}`, `)`, `]` en fin de ligne
    - après `}`

- Bizarrerie ???

```
var i = 3
if i <
3 {

}
```

Le langage Go

La portée des identificateurs

- Les portées définies par Go sont :
  - la portée globale
    - première lettre **majuscule** => **visible** de l'extérieur
  - la portée locale
- Go autorise le masquage

Le langage Go

Inférence de type, initialisation

- Go utilise l'inférence de type à l'initialisation

- Une variable non initialisée reçoit une valeur par défaut
  - `0` pour les numériques
  - `false` pour les booléens
  - `nil` pour les pointeurs

Le langage Go

Variable globale

- Une variable globale est nécessairement définie avec le mot-clé `var`
- L'opérateur d'initialisation est =
- L'initialisation peut-être faite sous la forme de tuple
- L'ordre d'initialisation est de la gauche vers la droite

```
var v1=2, v2=333*v1, v3=(v1==v2)
```

```
var x, y = 12, 24
```

```
var x1, x2, x3 = x1+x2+1, 0, 1
```

Le langage Go

Variable globale

- Pour typer une variable, le type doit suivre l'identificateur

```
var v int = 32
```

Le langage Go

Variable locale

- On peut utiliser la même syntaxe (`var`) pour une variable locale
- Mais il existe une forme courte avec `:=` (préférée) mais qui ne permet pas de typer la variable et oblige son initialisation (qui elle est typée)
  - Son typage est obtenu par inférence

```
v := 12
z := float64(13)
```

Le langage Go

Variable locale

- L'initialisation tuple fonctionne aussi

```
v, w := 12, 34
```

- Elle sert aussi d'affectation s'il y a au moins une variable nouvellement définie

```
v := 12
w, v := v, 34
```

- Attention, il n'est pas possible de faire référence aux variables en cours d'initialisation

```
x1, x2 := x2, 0 // interdit
```

Le langage Go

Affectation

- L'opérateur d'affectation (=) est tuple

```
v, w = 12, 34
```

- Les valeurs en partie droite sont calculées avant l'affectation

```
v, w := 1, 2
v, w = v+1, w+v+1
```

Le langage Go

Le `for`

- La boucle `for` a 4 formes
  - `for` *`definitions; test de continuation; instruction`* `{ ... }`
  - `for` *`test de continuation`* `{ ... }`
  - `for { ... }`
  - Le for d'itération avec `range` (voir plus loin)
- Attention, on ne doit pas parenthéser
  - `for i:=0; i<10; i++ { ... }`

Le langage Go

Le `for`

- Les définitions sont nécessairement des formes courtes (`:=`)
- Les variables définies sont dans la portée de la boucle

Le langage Go

Le `if`

- Deux formes de base
  - `if definitions; test { ... }`
  - `if test { ... }`
- puis les variantes avec `else`
- La portée des variables définies est celle du if et du else...

```
if i:=666; j<i {
} else {
}
```

Le langage Go

label, break, continue

- On peut étiqueter n'importe quel bloc d'instructions

```
unBloc : { ... }
```

- On peut sortir d'un for étiqueté ou non

```
unBloc : for ... { ... break unBloc; ... }
```

- On peut forcer la continuation d'un for étiqueté ou non

```
unBloc : for ... { ... continue unBloc; ... }
```

Le langage Go

label, goto

- On peut forcer à sauter vers un bloc
  - mais le saut ne doit pas comporter de définition de variable !
  - et on ne peut sauter à l'intérieur d'un bloc...

```
unBloc : { ... }
...
{ ... goto unBloc; ... }
```

Exercice

- La suite de Syracuse du nombre N est définie ainsi :
  - $S_0 = N$
  - $S_{n+1} = S_n/2$, si n est pair
  - $S_{n+1} = 3S_n+1$, si n est impair
- On définit
  - le vol comme la plus petite sous-suite qui termine par le nombre 1
  - la durée de vol comme la longueur de la sous-suite
  - l'altitude maximale comme le plus grand élément de la sous-suite

Exercice

- Écrire un programme qui pour une valeur de N donnée affiche:
  - le vol
  - la durée
  - l'altitude maximale
- Modifier le programme de sorte que ces données soient affichées pour tout $1 \leq N \leq 100$

Le langage Go

Le `switch`

- Il a deux formes possibles

```
switch définitions; sélecteur {
    case expression: ...
    ...
}
```

```
switch { // blank switch
    case expression: ...
    ...
}
```

- `default` peut apparaitre n'importe où
- les cas sont exclusifs, sinon il faut utiliser `fallthrough`

Le langage Go

Le `switch`

- Le switch est dynamique
  - La valeur des cas est calculée à l'exécution

```
switch v:=10; i {
    case 2*v+1 : ...
    ...
}
```

  - On peut y appeler des fonctions... (voir suite)
- Le blank switch...

Le langage Go

Les fonctions

- Une fonction est introduite avec le mot-clé `func`
  - elle peut avoir autant d'arguments que l'on souhaite
  - elle peut renvoyer autant d'arguments que l'on souhaite
- Le dernier argument peut être variadique (voir slices plus loin)

```
func f(i int) (int, int) {
    return i+1, i+2
}

func g() { ... }
```

Le langage Go

Les fonctions

- On peut nommer les arguments de sortie

```
func f(i int) (v1 int, v2 int) {
    v1, v2 = i+1, i+2
    return
}
```

- Il n'y a pas de surchage en Go...
- Les arguments sont passés par copie
- On peut définir des arguments non nommés (_)

Le langage Go

Les fonctions

- On peut définir des lambdas et faire de la programmation fonctionnelle

```
func f(i int) (v1 int, v2 int) {
    incroyable := 1
    g := func(n int) { return n+incr }
    v1, v2 = g(i), g(g(i))
    return
}
```

Exercice

- Cette fois on crée un module `syracuse` contenant deux fonctions
  - `CalculVol(n int)` qui renvoie la durée et l'altitude maximale
  - `AfficheVol(d int, a int)` qui affiche la durée et l'altitude maximale
- Le `main` appellera ces deux fonctions pour les valeurs $1 \leq N \leq 100$

Le langage Go

Quelques fonctions utiles...

- Pour tirer un nombre aléatoire entre 0 et N

```
import "math/rand"

rand.Intn(N)
```

- Pour lire au clavier

```
import "fmt"

var essai int

fmt.Scan(&essai)
```

Exercice

- Implémenter le jeu consistant à deviner un nombre tiré aléatoirement entre 1 et 100.

Le langage Go

Retour sur le `switch`

- Il existe une forme de `switch` appelée le blank switch ou switch sans condition
  - Il ne contient pas de clause de test
  - Il évalue les cas dans l'ordre et s'arrête au premier qui renvoie `true`

```
switch {
    case f(1,2): ...
    case rand.Int()<17: ...
    default: ...
}
```

- Il remplace agréablement une série de `if-else`

Le langage Go

La fonction `init`

- Toute fonction init définie dans un package sera automatiquement exécutée une et une seule fois préalablement à l'exécution du main
- Il peut y avoir autant de fonctions init que l'on souhaite dans un fichier/package.
  - dans un même fichier, elles seront exécutées dans l'ordre d'apparition
  - dans plusieurs fichiers (même package) c'est normalement l'ordre lexicographique des noms de fichiers (ordre par défaut des outils)

Le langage Go

La fonction `init`

- Toute fonction init définie dans un package sera automatiquement exécutée une et une seule fois préalablement à l'exécution du main
- Il peut y avoir autant de fonctions init que l'on souhaite dans un fichier/package.
  - dans un même fichier, elles seront exécutées dans l'ordre d'apparition
  - dans plusieurs fichiers (même package) c'est normalement l'ordre lexicographique des noms de fichiers (ordre par défaut des outils)

Le langage Go

Les constantes

- Le mot const permet de déclarer des constantes (globales ou locales)

```
const (
    PI = 3.141
    E = 2.718
)
```

Le langage Go

`iota`

- L'identificateur «magique» `iota` sert à simplifier la définition des constantes
  - Pour chaque utilisation de `const`, `iota` est initialisé à 0
  - À chaque utilisation de iota elle est incrémentée de 1

```
const (
    LUNDI = iota
    MARDI = iota
    MERCREDI = iota
)
```

```
const (
    JANVIER = iota
    FEVRIER = iota
    MARS = iota
)
```

```
const (
    _ = iota
    UN = iota
    DEUX = iota
    TROIS = iota
    CINQ = iota+1
    X = 2*iota
)
```

Le langage Go

Les types de base

- Entiers : `int`, `int8`, `int16`, `int32`, `int64` (|`int`|≥32 bits)
- Entiers non signés : `uint`, `uint8`, `uint16`, `uint32`, `uint64`
- Flottants : `float32`, `float64` (IEEE 754)
- Complexes : `complex64`, `complex128` (ex.: 5+3i)
- Booléens : `bool`
- Caractère : `rune`
- Chaîne de caractères : `string`
- Autres : `byte`, `error`

Le langage Go

Les types de base

- Aucune conversion implicite n'existe entre types de base !!!
- Il faut utiliser une conversion explicite

```
var i int = 666
var i32 int32

i32 = int32(i)
```

Le langage Go

Les types de base et littéraux

```
12 // décimal
012 // octal
0x12 // hexadécimal ou OX12
0b10101010_10101010 // binaire

1.5e16 // flottant...


true, false

'A' '😀' '\U0001F600'

"Bonjour"
```

Le langage Go

Les pointeurs

- Ils fonctionnent comme en C mais la syntaxe diffère

```
var p *int
```

  - Mais l'échappement est possible !

```
func f() *int {
    i := 666
    return &i
}
```

    - la variable sera créée dans le tas et non la pile

- Opérateur `new` (voir plus loin)

Le langage Go

Les tableaux

- Ils sont nécessairement dimensionnés (attention, si pas de taille ce sont des slices!!!! voir plus loin)

```
var t = [50]int {}
var t2 = [12]int { 1, 2, 3 }
var t3 = [...]int { 1, 2, 3 }
var t4 = [10]int { 5:22, 4:21 }
```

- La taille est obtenue par la fonction `len()`

```
lg := len(t2)
```

- Opérateurs `[]`, `==`, `!=`, `&`

Le langage Go

Les slices

- Un slice est un « tableau » dynamique (taille dynamique + capacité)
  - Le type d'un slice
  ```
  var s []int
  ```
  - La création d'un slice de longueur 20
  ```
  s = make([]int,20)
  ```
  - La taille est obtenue par la fonction `len()`
  ```
  lg := len(s)
  ```
  - L'opérateur [] ne fonctionne que sur les éléments existants !
  - On peut aussi créer un slice initialisé
  ```
  s := []int { 1, 2, 3 }
  ```

Le langage Go

Les slices

- Pour ajouter un(des) élément(s)

```
s := append(s, 1, 2, 3)
```

  - Toujours récupérer le retour de append
  - Le type slice se manipule comme une sorte de pointeur sur un tableau qui contient les éléments et append peut faire de la réallocation pour agrandir le tableau...

Le langage Go

Les slices

- On peut créer un slice à partir d'un tableau

```
t := [...]int { 1, 2, 3, 4, 5 }
s := t[2:4]
```

- Attention, le slice utilise le tableau comme stockage sous-jacent (sauf en cas d'utilisation de append)

```
s[0] = 111
fmt.Println(t, s)
s = append(s, 9, 8)
fmt.Println(t, s)
s[0] = 222
fmt.Println(t, s)
```

Le langage Go

Les slices

- Pour supprimer un élément on peut écrire

```
s = append(s[:i],s[i+1:]...)
```

- Il existe le package `slices` contenant différentes fonctions utiles de manipulation des slices
  - dont une fonction pour supprimer des éléments...

Exercice

- Reprendre «syracuse»
- Cette fois on modifie la suite de sorte que étant donné K impair on calcule
  - $S_{n+1} = S_n/2$, si n est pair
  - $S_{n+1} = 3S_n+K$, si n est impair
- La suite « boucle » mais on ne sait pas exactement comment (pas toujours sur 4, 2, 1)
  - Il faut donc détecter les cycles en employant un slice des valeurs atteintes au fur et à mesure...