Les bases du langage Go Partie 2 : pointeurs, tableaux, arguments

Initiation au développement

BUT informatique, première année

Ce TP a pour but de vous présenter les bases du langage Go. C'est ce langage qui sera utilisé pour réaliser tous les exercices de ce cours d'initiation au développement. Il n'y a pas de prérequis particuliers en dehors du contenu du TP précédent (les bases du langage Go partie 1).

Dans tout ce TP un texte avec cet encadrement est un texte contenant une information importante, il faut donc le lire avec attention et veiller à en tenir compte.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

Même si la plupart des exercices de ce TP peuvent vous sembler très faciles, faites les tous sérieusement : cela vous permettra de retenir la syntaxe du langage Go et d'être ainsi plus à l'aise lors des prochains TPs.

Si vous souhaitez avoir un autre point de vue sur l'apprentissage du langage Go, vous pouvez consulter le tutoriel officiel *A Tour of Go*[1] ou le site *Go by Example*[2] en faisant toute fois attention au fait que ces sites sont plutôt faits pour les personnes ayant déjà une bonne expérience de la programmation.

1. `https://tour.golang.org/`
2. `https://gobyexample.com/`

1 Les pointeurs

Les pointeurs sont des variables avec un type particulier et qui permettent de stocker l'endroit ou des valeurs sont situées en mémoire (leur adresse mémoire) plutôt que directement ces valeurs. Nous allons voir comment fonctionnent les pointeurs en Go et à quoi ils peuvent servir.

1.1 Passage d'arguments par référence et passage d'arguments par valeur

Si on considère une fonction déclarée par func f(x int, y int), on appelle x et y les *arguments* de cette fonction. Dans un langage de programmation qui offre la possibilité d'avoir des fonctions (comme le Go) on peut imaginer essentiellement deux mécanismes pour donner des arguments lors d'un appel de fonction (on parle de *passer* les arguments à la fonction). Pour simplifier la présentation de ces mécanismes, on peut considérer que chaque variable d'un programme est une boîte qui contient une valeur.

Passage d'arguments par valeur. Si les arguments sont passés *par valeur* cela signifie que lorsqu'on appelle notre

fonction f en écrivant f(n, m) ce sont les valeurs contenues dans les boîtes n et m qui vont être utilisées par la fonction et pas les boîtes elles-mêmes.

Ceci est sans doute plus clair sur un exemple. Si le passage des arguments se fait par valeur, alors à la fin du programme 1 la valeur de n n'aura pas changé : appeler f(n, m) ou f(1, 2) ne fait aucune différence.

```
package main

func f(x int, y int) {
    x = x + y
}

func main() {
    var n int = 1
    var m int = 2
    f(n, m)
}
```

Programme 1 – valeur-reference.go

Passage d'arguments par référence. Si les arguments sont passés par *référence*, lors d'un appel à f(n, m) on passe à la fonction directement les boîtes n et m. Et donc cette fonction peut modifier le contenu des boîtes, ce qui lui permet de changer la valeur d'une variable qu'on lui

passe en argument. Dans ce cas, à la fin du programme 1 la variable n aura changé, elle contiendra maintenant la valeur 3.

> Proposez un moyen de déterminer si, en Go, le passage d'arguments se fait par valeur ou par référence (vous pouvez pour cela récupérer le fichier valeur-reference.go sur MADOC) et mettez-le en œuvre.

1.2 Pointeurs

Le passage d'arguments par référence est quand même parfois utile. Pour y avoir accès on utilise ce qu'on appelle des *pointeurs*.

Pour représenter les pointeurs on utilise le symbole *, qu'on peut associer à n'importe quel type. Ainsi, par exemple, *int est le type d'un pointeur vers un int et *bool celui d'un pointeur vers un bool. Si on reprend l'analogie des boîtes, une variable de type int est une boîte qui contient un entier alors qu'une variable de type *int est une boîte qui contient une boîte qui elle-même contient un entier.

Si n est une variable vous savez accéder à son contenu, il suffit d'écrire son nom. Mais, si on souhaite accéder à l'entier correspondant à a de type *int c'est un peu plus compliqué : il faut récupérer la valeur qui est dans la boîte

contenue dans a. Pour cela on écrit *a et on dit qu'on déréférence le pointeur a. Parfois, on peut aussi avoir besoin d'accéder à la boîte correspondant à une variable n, par exemple pour la mettre dans une autre boîte. Pour cela on peut écrire &n.

Le programme 2 donne quelques exemples d'utilisation.

```go
package main

import "fmt"

func main() {
	var n int = 5
	var a *int = &n
	fmt.Println(n, *a)

	n = 6
	fmt.Println(n, *a)

	var m int = 7
	a = &m
	fmt.Println(n, *a, m)
}
```

Programme 2 – pointeurs.go

Récupérez le fichier pointeurs.go sur MADOC et exécutez-le. En utilisant l'analogie des boîtes présentée plus haut, représentez ce qu'il se passe lors de l'exécution de ce programme.

Quelle est la valeur par défaut d'une variable de type *int ? Que se passe-t-il quand on essaye de déréférencer un tel pointeur ? Expliquez cela à l'aide de l'analogie des boîtes.

Si on a une variable a de type *int on peut bien sûr aussi modifier le valeur contenue dans la boîte contenue dans a. Il suffit pour cela d'utiliser la notation *a, permettant d'accéder à cette valeur, à gauche d'un signe =, par exemple *a = 5.

Modifiez le fichier pointeurs.go pour, après le programme actuel, mettre la valeur 12 dans la boîte contenue dans a puis afficher la valeur contenue dans cette boîte. Testez-votre programme. Affichez après cela la valeur contenue dans m. Que remarquez-vous ? Expliquez cette observation à l'aide de l'analogie des boîtes.

Puisque on peut avoir un pointeur vers un objet de n'importe quel type, on peut aussi avoir un pointeur vers un

pointeur (et un pointeur vers un pointeur vers un pointeur, etc). Il suffit pour cela de rajouter une étoile devant : **int est une boîte qui contient une boîte, qui contient une boîte, qui contient un entier. Le programme 3 donne un exemple de cela.

```go
package main

import "fmt"

func main() {
    var n int = 5
    var a *int = &n
    var b **int = &a
    var c ***int = &b
    fmt.Println(n, *a, **b, ***c)

    ***c = 7
    fmt.Println(n, *a, **b, ***c)
}
```

Programme 3 – pointeurs2.go

Récupérez le ficher pointeurs2.go sur MADOC puis exécutez-le. Expliquez ce que vous observez à l'aide de l'analogie des boîtes.

L'analogie des boîtes a ses limites et n'est pas toujours totalement intuitive : il est tout à fait possible d'avoir deux pointeurs qui indiquent la même adresse mémoire, c'est-à-dire deux boîtes qui contiennent la même boîte.

1.3 Pointeurs et passage d'arguments par référence

Une utilité importante des pointeurs est de permettre de simuler le passage d'arguments par référence. Dans le programme 4 l'effet de la fonction fval sur n est exactement le même que celui de la fonction fref sur m. Cependant, cette deuxième tire profit des pointeurs pour simuler un passage d'arguments par référence.

Récupérez le fichier valeur-reference2.go sur MADOC et exécutez-le. Ajoutez éventuellement des affichage (fmt.Println) pour mieux voir ce qui se passe. Expliquez ce qui se passe lors des deux appels de fonctions en utilisant l'analogie des boîtes.

```go
package main

func fval(x int) int {
    return 2*x + 1
}

func fref(a *int) {
    *a = 2*(*a) + 1
}

func main() {
    var n int = 5
    n = fval(n)

    var m int = 5
    fref(&m)
}
```

Programme 4 – valeur-reference2.go

Récupérez le fichier unefonction.go sur MADOC, lisez son code puis exécutez-le. Modifiez la fonction f et ses appels de façon à simuler le passage d'arguments par référence (sur le modèle de l'exemple précédent avec fval et fref). Testez votre code pour vérifier que les affichages restent les mêmes.

Déréférencer un pointeur à un coût supérieur au simple accès à la valeur d'une variable. Cependant, passer les arguments par valeurs a aussi un coût (celui de la copie des valeurs) qui peut être important pour certains types un peu complexes (nous verrons leur utilité et comment en définir soit-même plus tard).

2 Retour sur les tableaux

Écrivez une fonction qui prend en argument un tableau d'entiers et ajoute 1 à chaque valeur contenue dans celui-ci. Testez votre fonction sur plusieurs tableaux.

La fonction que vous avez écrite prend en argument un tableau d'entiers et pas un pointeur vers un tableau d'entiers. Pourtant, les valeurs contenues dans le tableau sont

bien modifiées. Cela veut dire que le passage de l'argument s'est fait par référence. Nous allons maintenant essayer de bien comprendre cela.

2.1 Tableaux, *slices* et pointeurs

En réalité, en Go, il existe deux types de tableaux. Nous utiliserons leurs noms anglais pour les distinguer : *array* et *slice*.

Array. Un *array* est un tableau de taille fixe, sa taille faisant partie intégrale de son type. Ainsi, nous avons des tableaux d'entiers de longueur 1, 2, 3, etc, qui sont des types différents, qu'on note [1]int, [2]int, [3]int, etc. Un tel tableau de taille n correspond juste à n espaces mémoire contigus de la taille d'un élément du tableau. Ces tableaux sont des valeurs normales et sont donc passés par valeur aux fonctions, comme on peut le voir dans le programme 5.

Récupérez le fichier array.go sur MADOC et exécutez-le. Constatez que le tableau tab n'est pas modifié par la fonction f. Modifiez le code en utilisant un pointeur pour faire en sorte que la fonction f modifie tab.

```go
package main

import "fmt"

func f(t [4]int) {
    for i := 0; i < len(t); i++ {
        t[i] = 0
    }
}

func main() {
    var tab [4]int = [4]int{1, 2, 3, 4}
    fmt.Println(tab)
    f(tab)
    fmt.Println(tab)
}
```

Programme 5 – array.go

Concevez un programme permettant de savoir quelle est la valeur par défaut d'une variable de type [5]int, d'une variable de type [3]string et d'une variable de type [1]bool. Testez-le.

Slice. Les tableaux que nous avons utilisés dans le précédent sujet sont des tableaux de taille variable, qu'on appelle *slice* en anglais. Il n'existe qu'un seul type de *slice* d'entiers, qu'on note []int. Un *slice* est constitué de trois éléments : un entier indiquant sa taille, un entier indiquant sa capacité (nous reviendrons là dessus un peu plus tard) et un pointeur vers un *array*. Quand on accède à une case d'un *slice* c'est donc légèrement différent de quand on accède à une case d'un *array* : il y a un déréférencement qui a lieu. Ainsi, dans le programme 6 la fonction f modifie vraiment le tableau tab.

Récupérez le fichier slice.go sur MADOC et exécutez-le. Constatez que le tableau tab est modifié par la fonction f.

Il est assez rare qu'on utilise les *array* en Go, à partir de maintenant nous nous concentrerons sur les *slices*, que nous appellerons simplement tableaux.

```go
package main

import "fmt"

func f(t []int) {
    for i := 0; i < len(t); i++ {
        t[i] = 0
    }
}

func main() {
    var tab []int = []int{1, 2, 3, 4}
    fmt.Println(tab)
    f(tab)
    fmt.Println(tab)
}
```

Programme 6 – slice.go

2.2 Découpages

Il est possible de récupérer une partie d'un tableau et de la stocker dans un autre tableau. Ceci se fait en écrivant entre crochets, séparés par le symbole :, les indices de début et de fin (exclu) de la partie de tableau qu'on souhaite couper. Le programme 7 montre quelque exemples de cette opération.

```go
package main

import "fmt"

func main() {
    var tab []int = []int{2, 12, 22, 11, 3, 5,
    fmt.Println(tab)

    var t1 []int = tab[3:7]
    fmt.Println(t1)

    var t2 []int = tab[:7]
    fmt.Println(t2)

    var t3 []int = tab[3:]
    fmt.Println(t3)
}
```

Programme 7 – slice2.go

Récupérez le fichier slice2.go sur MADOC et exécutez-le. Que se passe-t-il quand on ne met pas d'indice de début ? Que se passe-t-il quand on ne met pas d'indice de fin ?

Récupérez le fichier slice3.go sur MADOC et exécutez-le. Expliquez à l'aide de l'analogie des boîtes ce que vous observez.

Attention, quand on découpe des tableaux le résultat qu'on obtient est stocké en mémoire au même endroit que le tableau de départ (plus précisément, les pointeurs des deux tableaux indique le même tableau — *array* — sous-jacent). Si l'on modifie l'un, cela modifie aussi l'autre.

2.3 Un peu plus sur les chaînes de caractères

Les chaînes de caractères (string) se manipulent comme des tableaux. On peut accéder à un caractère d'une chaîne s par son indice, par exemple s[2] pour le troisième caractère. On peut donc aussi découper les chaînes de caractères avec la syntaxe vue juste au dessus.

Écrivez une fonction qui prend en argument une chaîne de caractères constituée de deux mots séparés par un espace et retourne deux chaînes de caractères, chacune contenant l'un des deux mots. Afin de vérifier si un caractère d'une chaîne est un espace on peut le comparer à ' ', par exemple s[2] == ' ' vaut true si le troisième caractère de s est un espace et false sinon. Testez votre fonction sur plusieurs exemples.

2.4 Création de tableaux

Jusqu'à présent, on a toujours créé nos tableaux en indiquant directement leur contenu. Cependant, on peut parfois vouloir simplement créer un tableau d'une taille donnée, sans pour autant spécifier ce qui va dedans, par exemple quand on va remplir ce tableau par la suite en fonction de résultats de calculs. Le programme 8 montre quelques exemples.

Dans ce programme on utilise make, qui prend deux arguments, pour créer nos tableaux. Le premier argument qu'on doit donner est le type du tableau à créer ([]int, []bool, []string, etc). Le deuxième argument est un entier qui sert à indiquer la taille du tableaux qu'on veut créer.

```go
package main

import "fmt"

func main() {
    var itab []int = make([]int, 5)
    var btab []bool = make([]bool, 3)
    fmt.Println(itab, btab)
}
```

Programme 8 – slice4.go

Récupérez le fichier slice4.go sur MADOC et exécutez-le. Quelles sont les valeurs contenues dans un tableau créé à l'aide de make ?

Écrivez une fonction qui prend en argument un entier n, crée un tableau contenant les n premiers multiples de 7, et retourne ce tableau. Testez-votre fonction pour plusieurs valeurs de n (0, 5, 100, etc).

2.5 Agrandissement et capacité

On a appelé les *slices* tableaux de taille variable, mais, pour le moment, on n'a vu que le moyen de réduire leur taille en les découpant. Il est aussi possible d'augmenter

la taille d'un tableau. Ceci se fait en utilisant la fonction append.

La fonction append. Cette fonction prend en arguments un tableau et un élément et retourne un tableau correspondant au tableau donné en argument à la suite duquel l'élément donné en argument a été ajouté. Le programme 9 montre un exemple.

```
package main

import "fmt"

func main() {
    var tab []int = make([]int, 5)
    fmt.Println(tab, len(tab))
    tab = append(tab, 2)
    fmt.Println(tab, len(tab))
}
```

Programme 9 – slice5.go

Récupérez le fichier slice5.go sur MADOC et exécutez-le.

La capacité. On a indiqué précédemment qu'un tableau possède, en plus de sa taille, une capacité. Celle-ci cor-

respond à une quantité d'espace réservée pour stocker le tableau : la taille du tableau est son nombre d'éléments, sa capacité est son nombre d'éléments potentiels. On peut voir la capacité comme la taille de l'*array* sous-jacent. Elle peut se définir en ajoutant un troisième argument à make lorsque l'on crée un tableau, et on peut la connaître en utilisant la fonction cap. Le programme 10 montre quelques exemples.

```go
package main

import "fmt"

func main() {
    var tab []int = make([]int, 3, 5)
    fmt.Println(tab, len(tab), cap(tab))

    var tab2 []int = make([]int, 3)
    fmt.Println(tab2, len(tab2), cap(tab2))

    var tab3 []int = []int{0, 0, 0}
    fmt.Println(tab3, len(tab3), cap(tab3))
}
```

Programme 10 – slice6.go

Récupérez le fichier slice6.go sur MADOC et exécutez-le. Quelle est la capacité d'un tableau créé par make lorsqu'on n'indique que la taille de ce tableau ? Quelle est-elle lorsqu'on crée le tableau sans utiliser make ?

Lien entre append et capacité Lorsqu'on utilise append le résultat va varier en fonction de la capacité : si celle-ci est suffisante, l'élément sera ajouté dans l'*array* sous-jacent, c'est-à-dire au même endroit dans la mémoire que le tableau d'origine. Par contre, si la capacité est insuffisante, le tableau d'origine sera copié dans un espace plus grand, puis l'élément sera ajouté à la suite. Le programme 11 permet de mieux comprendre cela.

Récupérez le fichier slice7.go sur MADOC et exécutez-le.

Au début, on peut observer que lorsqu'on modifie la première case de t cela change aussi le contenu de la première case de tab. En effet, ces deux tableaux correspondent au même espace en mémoire (ils ont le même tableau sous-jacent). On remarque ensuite que, au tour de boucle 4, la taille de t devient égale à sa capacité. Au tour suivant la capacité de t a augmenté : le tableau a été

```go
package main

import "fmt"

func main() {
	var length int = 3
	var capacity int = 5
	var tab []int = make([]int, length, capacity)
	var t []int = tab
	var max int = 5
	for i := length; i <= max; i++ {
		t = append(t, i)
		t[0] = i
		fmt.Println("Tour de boucle", i)
		fmt.Println("\ttab =", tab, "len(tab) =", len(ta
		fmt.Println("\tt =", t, "len(t) =", len(t), "cap
	}
}
```

Programme 11 – slice7.go

copié dans un espace plus grand. On voit que c'est bien une copie car à partir de ce moment, modifier la première case de t n'affecte plus la première case de tab.

Testez à nouveau le programme du fichier slice7.go en modifiant les valeurs de length, capacity et max. Essayez d'en déduire une règle sur la façon dont la capacité d'un tableau est augmentée lorsque c'est nécessaire.

Le fait de copier un tableau a un coût, c'est pour essayer d'éviter au maximum les copies quand on ajoute des éléments qu'on permet de réserver un espace mémoire plus grand que nécessaire pour nos tableaux (grâce à la capacité).

Lien entre découpages et capacité Lorsqu'on découpe un tableau, comme on l'a vu plus haut, on peut se demander quelle sera sa capacité. Ceci peut se déterminer expérimentalement.

Récupérez le fichier slice8.go sur MADOC. Exécutez-le pour différentes valeurs des variables length, capacity, start et end et déduisez-en comment sont calculées la taille et la capacité d'un découpage de tableau. Testez aussi des valeurs de end qui dépasse la taille ou même la capacité de tab.

2.6 Copies de tableaux

On l'a vu dans les parties précédentes : les tableaux peuvent s'avérer complexes et il est parfois difficile de savoir si deux tableaux partagent un même espace mémoire (et donc si modifier l'un affectera l'autre) ou pas.

Lorsqu'on a besoin de récupérer tout ou partie d'un tableau dans un autre, en s'assurant que les deux tableaux sont indépendants, on peut utiliser la fonction copy. Le programme 12 donne un exemple d'utilisation de cette fonction.

On peut observer que la fonction copy s'appelle avec deux arguments. Le premier est la destination : l'endroit où on va placer la copie. Le deuxième est la source : le tableau qui doit être copié.

```go
package main

import "fmt"

func main() {
    var tab []int = []int{1}
    var t []int = make([]int, 1)
    copy(t, tab)
    fmt.Println("tab:", tab, len(tab), cap(tab
    fmt.Println("t:", t, len(t), cap(t))

    t[0] = 0
    fmt.Println("tab:", tab, len(tab), cap(tab
    fmt.Println("t:", t, len(t), cap(t))
}
```

Programme 12 – slice9.go

Récupérez le fichier slice9.go sur MADOC et exécutez-le.

En testant, on se rend compte que lorsqu'on modifie la première case du tableau t cela n'a pas d'effet sur le tableau tab. On à bien réalisé une copie.

Dans le fichier slice9.go faites varier la taille et la capacité du tableau t et la taille du tableau tab. Que se passe-t-il quand la taille de t est inférieure à celle de tab ? Que se passe-t-il quand la capacité de t est inférieure à la taille de tab ?

2.7 Tableaux de tableaux (de tableaux de tableaux. . .)

Puisqu'on peut créer un tableau d'éléments de n'importe quel type, on peut aussi créer des tableaux de tableaux. Le programme 13 donne un exemple.

Dans ce programme on observe tout d'abord la création d'un tableau de tableaux d'entiers. Ici on dit explicitement le contenu de notre tableau. Son type est [][]int donc sont contenu est écrit sous la forme [][]int{contenu}. Les éléments qu'il contient sont des tableaux d'entiers (type []int) donc chacun d'entre eux est déclaré exactement comme

```go
package main

import "fmt"

func main() {
    var tab [][]int = [][]int{
        []int{1, 2, 3},
        []int{4, 5},
        []int{6, 7, 8, 9, 10},
    }

    fmt.Println("tab =", mat)
    fmt.Println("tab[0] =", mat[0])
    fmt.Println("tab[0][1] =", mat[0][1])
}
```

Programme 13 – tabtab.go

on l'a vu avant : []int{contenu}.

Ensuite, on voit comment accéder aux éléments de ce tableau : tab fait référence à tout le tableau, tab[i] fait référence au tableau d'entiers numéro i dans le tableau de tableaux, et tab[i][j] fait référence à l'entier numéro j de ce tableau numéro i.

Récupérez le fichier tabtab.go sur MADOC et exécutez-le.

On peut bien sûr utiliser make pour créer des tableaux de tableaux. Il faudra alors bien penser à initialiser (par exemple avec make), si nécessaire, chacun des tableaux de notre tableau de tableaux.

Écrivez un programme qui initialise un tableau t de n tableaux de taille m, puis qui le rempli de telle sorte que t[i][j] soit égal à $i \times j$. On construit donc un tableau contenant les tables de multiplication de 0 à n-1, chacune rangée dans un tableau. Testez votre programme pour différentes valeurs de n et m.

Vous pouvez trouver une très bonne explication, plus détaillée, du fonctionnement interne des tableaux en Go sur le site go.dev [3]

3. `https://go.dev/blog/slices-intro`

3 Arguments de la ligne de commande

Dans les exercices précédents on a souvent testé des programmes pour différentes valeurs de variables. Pour cela, il fallait à chaque fois modifier le code du programme puis utiliser la commande go run (qui elle-même compile le fichier et l'exécute). Ce n'est pas forcément raisonnable de faire cela : modifier le code à chaque fois est source d'erreurs et la compilation a un coût (en énergie et en temps, faible pour les programmes que vous avez écrits jusqu'à présent, mais qui peut être beaucoup plus élevé dès que les programmes deviennent plus complexes).

Nous allons voir dans cette partie comment il est possible de modifier les valeurs de certaines variables au moment de l'exécution d'un programme : sans avoir à la recompiler et sans avoir à modifier le code source.

3.1 Séparer la compilation de l'exécution

Quand on exécute un programme prog.go en utilisant la commande go run prog.go cela donne en réalité lieu (de manière transparente) à deux étapes : la compilation, qui permet de produire un fichier exécutable à partir du fichier source, puis l'exécution qui permet de faire fonctionner le programme.

Parfois on souhaite exécuter un programme plusieurs fois, sans avoir modifié son code. Il n'est alors pas raisonnable de le recompiler à chaque fois. On peut séparer la compilation de l'exécution en utilisant la commande go build à la place de go run.

Compilez le programme tabtab.go en utilisant la commande go build tabtab.go à la place de go run tabtab.go. Vous obtenez normalement un fichier exécutable nommé tabtab.

Le fichier exécutable obtenu par la compilation peut ensuite être utilisé en tapant dans le terminal un chemin vers ce fichier. Si vous vous situez dans le répertoire où est le fichier exécutable prog vous pouvez utiliser la commande ./prog pour l'exécuter.

Exécutez le programme tabtab créé précédemment et vérifiez que vous obtenez bien le même résultat qu'en faisant directement go run tabtab.go.

3.2 Définir des valeurs à l'exécution

Lorsque vous utilisez une commande dans un terminal, vous indiquez d'abord le nom de cette commande, puis un certain nombre d'*arguments* séparés par des es-

paces. Le résultat de la compilation d'un programme Go est une commande, vous pouvez donc lui donner des arguments. Ainsi, si j'ai construit un exécutable prog je peux utiliser la commande ./prog 12 bonjour pour lui donner les arguments 12 et *bonjour*. Il est ensuite possible pour le programme de récupérer et d'utiliser ces arguments. Pour cela on utilise la bibliothèque os (qu'il faudra penser à importer).

Le programme 14 donne un exemple de récupération d'arguments dans un programme Go à l'aide de la bilbiothèque os.

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Le programme s'appelle", os.A
    fmt.Println("Il y a", len(os.Args)-1, "arg
    for argPos := 1; argPos < len(os.Args); ar
        fmt.Println("L'argument", argPos, "vau
    }
}
```

Programme 14 – args.go

Une fois la bibilothèque os importée on a accès à un tableau os.Args qui contient le nom du programme exécuté, suivi des arguments donnés à ce programme. On peut donc obtenir le nombre d'arguments en regardant la longueur de ce tableau. Et on peut accéder aux arguments dans les cases du tableau.

On notera que l'importation de plusieurs bibliothèque peut se faire en les listant toutes, une par ligne, entre parenthèse après le import.

Récupérez le fichier args.go et compilez-le avec go build. Testez le fichier exécutable obtenu (args) avec différents nombres et types d'arguments.

Exécutez le fichier args.go avec go run. Où se trouve le programme compilé par cette commande ?

Écrivez un programme qui stocke son premier argument dans une variable entière. Compilez-le avec go build. Que pouvez-vous déduire sur la nature des arguments récupérés par un programme ?

On dispose d'un certain nombre de fonctions utiles pour transformer des chaînes de caractères en d'autres types. Nous ne les verrons pas toutes ici, mais pour interpréter une chaîne de caractères comme un entier on peut uti-

liser strconv.Atoi qui provient de la bibliothèque strconv. Cette fonction prend en argument une chaîne de caractère et retourne un entier et une erreur. L'erreur sert à savoir s'il y a eu un problème durant la conversion (si elle vaut nil c'est que tout s'est bien passé). L'entier est l'interprétation de la chaîne de caractères sous forme d'entier. Pour plus de précisions, n'hésitez pas à consulter la documentation.

Modifiez votre programme précédent pour convertir votre argument en entier avant de le stocker dans votre variable (n'oubliez pas de récupérer l'erreur dans une autre variable). Compilez à nouveau votre programme (go build). Testez-le ensuite pour différentes valeurs d'argument (y compris des arguments qui ne sont pas des entiers), ainsi que sans argument. Analysez les résultats et les erreurs obtenues.

Tout ceci est sans doute un peu laborieux pour récupérer des arguments. Il existe une bibliothèque pour vous simplifier la vie : la bibliothèque flag.

3.3 La bibliothèque flag

La bibliothèque flag peut se charger pour vous de mettre les arguments d'un programme dans des variables, en s'assurant qu'ils ont le bon type et même en leur donnant une valeur par défaut. Le programme 15 donne un exemple d'utilisation de cette bibliothèque.

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    var n int
    flag.IntVar(&n, "setn", 27, "Fixe la valeu
    flag.Parse()
    fmt.Println("n vaut", n)
}
```

Programme 15 – flag.go

Dans ce programme on commence par déclarer un entier n.

Ensuite on utilise la fonction flag.IntVar qui prépare la lecture d'un argument de type entier. Cet argument sera stocké dans la variable n (on donne pour cela un pointeur

vers n en premier argument à la fonction). Il aura pour valeur par défaut 27 (troisième argument de la fonction). Dans la ligne de commande, pour fixer une valeur à cet argument il faudra écrire -setn=12 (on peut remplacer 12 par n'importe quelle autre valeur).

Enfin, la fonction flag.Parse() permet de prendre en compte toutes les lectures d'arguments qui ont été préparées avant (ici par flag.IntVar).

Récupérez le fichier flag.go sur MADOC et compilez-le avec go build. Essayez ensuite de l'exécuter de différentes façon (avec et sans argument, avec différents arguments).

Testez votre programme avec l'argument -h, c'est-à-dire en faisant ./flag -h. À quoi sert le quatrième argument de la fonction flag.IntVar ?

On peut bien sûr traiter d'autres types d'arguments que des entiers (avec les fonctions flag.XxxVar adéquates), plusieurs arguments à la fois (avec plusieurs fonctions flag.XxxVar avant le flag.Parse), etc. Toutes les informations sur la bilbiothèque sont disponibles dans la documentation [4].

4. `https://pkg.go.dev/flag`

Écrivez un programme qui prend en argument trois entiers x, y et z et stocke dans un tableau la table de multiplication de x à partir de $0 \times x$ jusqu'à $y \times x$ à part les valeurs qui sont multiples de z. Testez votre programme pour plusieurs valeurs de x, y et z (sans le recompiler). Si nécessaire, modifiez votre programme pour vous assurer que la taille de votre tableau est juste la bonne pour stocker vos valeurs. Vous devez être capable de justifier que c'est bien le cas. Testez-le. Si nécessaire, modifiez votre programme pour vous assurer qu'il n'y a jamais de copies de votre tableau qui sont réalisées au cours de son remplissage. À nouveau, vous devez être capable de justifier que c'est bien le cas. Testez-le.