Les bases du langage Go
Partie 5 : plus de fonctions

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 des TP précédents (les bases du langage Go partie 1, partie 2, partie 3 et partie 4).

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/`

Jusqu'à présent nous avons vu comment créer des fonctions, avec un nom, des paramètres dont les types sont bien définis, et des valeurs de retour dont les types sont aussi bien définis. Nous avons aussi vu comment utiliser (appeler) ces fonctions dans du code Go pour les faire calculer des résultats. En Go (et dans un certain nombre de langages de programmation), il est possible de faire plus : passer les fonctions en paramètre d'autres fonctions, retourner des fonctions en résultat d'autres fonctions. Ceci permet de créer simplement des fonctions modulaires, et donc de réutiliser le plus possible son code plutôt que de le dupliquer.

1 Fonctions anonymes

Avant de voir comment et pourquoi passer des fonctions en paramètres d'autres fonctions et retourner des fonctions en résultats d'autres fonctions, nous allons voir qu'on peut définir en Go des fonctions dites *anonymes*, c'est-à-dire des fonctions aux quelles on ne donne pas de nom.

```go
package main

import "log"

func main() {

    var x int = func(x, y int) int { return x + y }(2, 3)

    var f func(int, int) int = func(x, y int) int { return x - y }

    log.Print(x, f(2, 3))

}
```

Programme 1 – anonyme.go

Le programme 1 montre deux exemples de déclarations de fonctions anonymes. On peut observer qu'une telle déclaration se fait exactement comme une déclaration de fonction classique, simplement sans spécifier de nom pour la fonction.

La première fonction anonyme est immédiatement utilisée pour calculer une valeur (stockée dans la variable x). Pour cela, à la suite de la déclaration de la fonction on indique directement entre parenthèses les arguments avec les quels on souhaite l'appeler. Suite à ça la fonction «n'existe plus».

La deuxième fonction anonyme est stockée dans une variable f du bon type (donc du type de la fonction). Cela revient en fait exactement à avoir directement déclaré une fonction f comme on le fait d'habitude, et, concrètement, on utilisera rarement, voir jamais, cette syntaxe. Cependant, sur le même modèle, on pourrait avoir un type structuré dont l'un des champs est une fonction, qui peut donc être différente entre deux éléments de ce type (ce qui ne serait pas le cas pour une méthode s'appliquant aux éléments de ce type).

Récupérez le fichier anonyme.go sur MADOC et testez-le. Assurez-vous d'avoir bien compris tout ce qui est fait dans ce code, et en particulier la façon dont on indique le type d'une (variable de type) fonction.

En elles-mêmes, les fonctions anonymes ne sont pas très utiles (les exemples vus pour le moment consistent en une fonction qui n'est utilisée qu'une seule fois — elle pourrait être remplacée par son code — et une déclaration plus verbeuse que nécessaire d'une fonction classique), cependant on verra qu'elles nous servirons notamment lorsqu'on souhaitera passer des fonctions en paramètres d'autres fonctions.

2 Fonctions en paramètres de fonctions

En Go, les fonctions peuvent prendre en paramètres d'autres fonctions. Ceci peut permettre de construire des fonctions plus générales que ce qu'on a fait jusqu'à présent. On peut en voir un exemple dans le paquet sort de la bibliothèque standard de Go. La fonction Slice [3] de ce paquet permet de trier un tableau selon n'importe quel ordre, cet ordre étant défini par une fonction.

```go
package main

import (
    "log"
    "sort"
)

func main() {

    var t []int = []int{1, 3, 2, 0, -3, 27, 5}
    log.Print(t)

    sort.Slice(t, func(i, j int) bool { return t[i] < t[j] })
    log.Print(t)

}
```

Programme 2 – tri.go

Le programme 2 trie le tableau t en ordre croissant. Pour cela, la fonction sort.Slice est appelée sur ce tableau avec comme deuxième paramètre une fonction qui indique que si t[i] est plus petit que t[j] alors il doit être avant dans le tableau trié.

Récupérez le fichier tri.go sur MADOC et testez-le. Assurez-vous d'avoir bien compris tout ce qui est fait dans ce code, et en particulier la façon dont on utilise la fonction anonyme pour indiquer quel tri réaliser. Modifiez ce code pour que le tableau soit trié en ordre décroissant.

On pourrait aussi définir la fonction de comparaison d'éléments pour le tri de manière plus générale : elle pourrait comparer des éléments directement, et ne pas dépendre du tableau à trier. Cependant, pour des raisons techniques (pouvoir trier des tableaux de n'importe quel type d'éléments) c'est un autre choix qui a été fait par les développeurs du langage Go.

Pour écrire soit même une fonction qui prend en paramètre une autre fonction, il suffit d'utiliser le mot clé func en définissant le type d'un paramètre.

Le programme 3 déclare une fonction applique qui remplace chaque entier d'un tableau par le résultat d'une fonction f appliquée à cet entier. Ceci est ensuite utilisé pour doubler la valeur de tous les éléments d'un tableau.

Récupérez le fichier applique.go sur MADOC et testez-le. Assurez-vous d'avoir bien compris tout ce qui est fait dans ce code.

Écrivez un programme qui définit une fonction replie prenant en paramètre un tableau d'entiers t et une fonction f à deux arguments entiers et calculant f(f(...f(f(t[0],t[1]),t[2]) ...),t[n-1]) puis qui utilise cette fonction pour calculer la somme des entiers contenus dans un tableau. On pourra supposer que les tableaux manipulés auront toujours au moins deux éléments.

3. https://pkg.go.dev/sort#Slice

```go
package main

import "log"

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

func main() {

    var t []int = []int{1, 3, 2, 0, -3, 27, 5}
    log.Print(t)

    applique(t, func(x int) int { return 2 * x })
    log.Print(t)

}
```

Programme 3 – applique.go

3 Fonctions comme valeurs de retour de fonctions

Enfin, en Go, les fonctions peuvent retourner des fonctions. Ceci permet de créer des fonctions en fonction de paramètres, puis de les réutiliser.

```go
package main

import (
    "log"
    "sort"
)

func plusPetit(t []int) func(i, j int) bool {
    return func(i, j int) bool { return t[i] < t[j] }
}

func main() {

    var t []int = []int{1, 3, 2, 0, -3, 27, 5}
    log.Print(t)

    sort.Slice(t, plusPetit(t))
    log.Print(t)

}
```

Programme 4 – tri2.go

Le programme 4 est similaire au programme 2. Cependant, au lieu d'utiliser une fonction anonyme pour indiquer comment trier le tableau, on génère cette fonction à l'aide d'une autre fonction : plusPetit. Ainsi, si on souhaite trier un autre tableau de la même façon, on peut simplement appeler plusPetit sur ce nouveau tableau plutôt que de réécrire toute la fonction anonyme.

Récupérez le fichier applique.go sur MADOC et testez-le. Assurez-vous d'avoir bien compris tout ce qui est fait dans ce code. Modifiez le programme pour déclarer un deuxième tableau après t et triez ce deuxième tableau en utilisant la même méthode que pour trier t.

Ceci fonctionne car les tableaux sont basés sur des pointeurs (l'espace mémoire utilisé pour le contenu de t et le contenu de sa copie dans la fonction engendrée par plusPetit sont donc les mêmes) et car sort.Slice ne modifie pas la taille des tableaux (ce qui pourrait «séparer» les deux espaces mémoire, comme on l'a vu avec l'utilisation de append sur un tableau dont la capacité maximale était atteinte).

Écrire une fonction add qui calcule et retourne la somme de deux entiers. Écrire une fonction buildSpecificAdder qui prend en paramètre un entier n et retourne une fonction qui prend en paramètre un entier, lui ajoute n et le retourne (par exemple, buildSpecificAdder(10) retournera une fonction à un argument qui ajoute 10 à son argument). La fonction buildSpecificAdder fournira donc une version spéciale de add dont l'un des arguments est fixé.