Introduction à la programmation Go

Hugo Blanc - Université Lyon 1
LP ESSIR 2024-2025

November 02, 2024

« To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. »

– Edsger Dijkstra, *The Humble Programmer*, 1972

Table des matières

0.1. Pré-requis

- Avoir une machine GNU/Linux en état de fonctionnement.
- Avoir un utilisateur différent de `root` appartenant au groupe `sudo`.
- Avoir une connexion à Internet.

1. Introduction

Go (Golang) est un langage **compilé**, **fortement typé** et **concurrent** inspiré du langage C. Il a été créé par Robert Griesemer, Rob Pike (Plan9, UTF-8…) et Ken Thompson (UNIX, `grep`, `sh`, Plan9, UTF-8…) chez Google, et la première version a été publiée le 10 novembre 2009. Depuis sa création, Go se veut être un langage simple mais performant, facilitant la programmation à grande échelle et la collaboration.

Go a également envahi le domaine des technologies Cloud: Kubernetes, Docker, ArgoCD, Grafana… sont codés en Go.

Pourquoi choisir Go pour l'administration système ?

Évidemment, les langages de programmation sont comme les goûts musicaux: tout se discute, et chacun.e a ses préférences. Mais voici une liste non-exhaustive des avantages que je trouve à Go pour l'administration système:

- **Go est simple**. Go est un langage assez simple à prendre en main, y compris par des novices en programmation. Un.e débutant.e en programmation sera capable de déchiffrer et comprendre une *codebase* en Go sans trop de problèmes comparé à d'autres langages plus costauds (Rust, …). La spécification du langage° est également courte (50 pages) et compréhensible, contrairement à d'autres (la spécification Java fait + de 700 pages !).
- **Go est performant**. C'est un fait et cela a été *benchmarké*[1] à de nombreuses reprises. Dans certains cas, vous serez amené.es à devoir parser des fichiers de log, ou traiter des dumps de base de données par exemple. Savoir et pouvoir développer dans un langage performant est un réel avantage
- **Go est un langage compilé**. C'est notamment pour ça que les programmes sont si performants. Cela permet également, dans une moindre mesure, de faire des déploiements conteneurisés plus sûrs (images from scratch dans Docker par exemple). Et encore mieux:
- **Go est un langage *cross-compilé*** ! Ce qui veut dire que dans 95% des cas, vous pourrez compiler votre code pour toutes les architectures majeures (amd64, ARM, etc.) sans avoir à toucher à une seule ligne de code ! Un tel comportement facilite grandement les déploiements.
- **C'est un langage fortement typé**. Fini les magouilles de types qui font planter les programmes au *runtime*.
- **La documentation, l'outillage et la communauté sont excellent.es**. Le web est rempli de ressources pour apprendre et progresser en Go. Il existe de nombreuses documentations très complètes en ligne, de niveau débutant à confirmé.
- **La librairie standard est très complète**. De nombreux *packages* sont inclus dans la libraire standard, ce qui permet de faire des applications complètes sans devoir constamment s'appuyer sur des librairies tierces.
- et j'en passe…

Plus généralement, Go excelle du développement web orienté *back-end* à la création d'outils en ligne de commande, en passant par des programmes d'[*Infra As Code*] ou pour faire de la programmation embarquée.

[1] https://www.golinuxcloud.com/golang-performance/°

> **Note**
>
> Malheureusement, par manque de temps, nous ne pourrons pas couvrir tous les aspects du langage. Certains éléments clés pour maîtriser Go ne sont pas détaillés dans ce cours, et chacun.e devra les explorer de son propre chef.

1.1. Installation et premiers pas

Cette partie couvre l'installation de la toolchain Go, et notamment le compilateur. Puis nous découvrirons un Hello World° écrit en Go.

> **Note**
>
> Ce guide montre comment installer le compilateur Go dans un environnement **Linux AMD64**.

1. Allez sur https://go.dev/doc/install°
2. Appuyez sur **Download (1.X.X)**
3. Dans la liste des fichiers, choisissez **go1.X.X.linux-amd64.tar.gz**
4. Une fois le fichier récupéré, allez dans son dossier et lancez la commande:

```
$ rm -rf /usr/local/go
$ sudo tar -C /usr/local -xzf go1.X.X.linux-amd64.tar.gz
```

5. Ajoutez le dossier `/usr/local/go/bin` à votre PATH de manière permanente (dans votre `.bashrc` par exemple):

```
$ echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.bashrc
```

6. Testez que votre installation est fonctionnelle:

```
$ go version
go version go1.21.0 linux/amd64
```

Nous allons réaliser un programme qui affichera « Hello, World! »

- Ouvrez un terminal et rentrez dans un répertoire temporaire:

```
$ cd $(mktemp -d)
```

- Créez un fichier `main.go` :

```
$ touch main.go
```

- Éditez-le pour qu'il contienne le programme suivant:

```go
package main

import "fmt"

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

- Puis lancez-le:

```
$ go run main.go
Hello, World!
```

Vous venez d'écrire et d'exécuter votre premier programme en Go!

Nous avons vu plus tôt que Go est un langage **compilé**. Vous pouvez essayer de le compiler puis de l'exécuter avec les commandes suivantes:

```
$ go build main.go
$ ls
main* main.go
$ ./main
Hello, World!
```

> **Note**
>
> Exécuter un programme déjà compilé est bien plus rapide que de le lancer via `go run`:
>
> ```
> $ time ./main
> Hello, World!
>
> real  0m0.004s
> user  0m0.001s
> sys     0m0.004s
> $ time go run main.go
> Hello, World!
>
> real  0m0.070s
> user  0m0.100s
> sys     0m0.132s
> ```

1.2. Anatomie d'un Hello, World!

Reprenons le programme écrit plus tôt, et regardons son contenu:

```go
package main

import "fmt"

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

- `package main` est le nom du *package* auquel le fichier appartient. Le package `main` est le principal package d'un programme **exécutable**.
- `import "fmt"` permet d'importer une librairie appelée `fmt`. Nous pouvons donc utiliser toutes les fonctions exportées par cette librairie dans notre code.
- `func main() {}` permet la déclaration d'une fonction `main` qui ne prend ni ne retourne de valeurs. La fonction `main` est le point d'entrée des exécutables en Go.
- `fmt.Println("Hello, World!")` réalise un appel de la fonction `Println` du package `fmt`. Cette fonction prend en paramètre une chaîne de caractères, ici « Hello, World ».

> **Précisions sur les extraits de code**
>
> Afin de simplifier la lecture, seuls les élément en lien avec le thème étudié sont écrits dans les exemples de code. On considère évidemment que ces bouts de code ne sont qu'un morceau d'un code plus large contenant une fonction `main` ainsi qu'un package associé.

2. Types et variables

Une variable est un´élément qui permet de **stocker** une donnée d'un **certain type**. Il existe un grand nombre de types et il est possible d'en définir soi-même

2.1. Types

Les principaux types mis à disposition par le langage sont:

- `bool` : stocke un booléen (vrai ou faux).
- `string` : stocke une chaîne de caractères.
- `int` , `int8` , `int16` , `int32` , `int64` : stockent un entier signé codé sur `n` bits.
- `uint` , `uint8` , `uint16` , `uint32` , `uint64` : stockent un entier non-signé sur `n` bits.
- `byte` : alias pour `uint8` .
- `rune` : alias pour `int32` , représente un caractère Unicode.
- `float32` , `float64` : représentent des nombres flottants codés sur 32 ou 64 bits.

Il existe également des types un peu plus spéciaux:

- `error` : type qui représente toute valeur qui peut se décrire comme un `string` , généralement utilisé pour contenir des messages d'erreurs[2].
- `interface` : définit et décrit les méthodes exactes qu'un autre type doit avoir.

2.2. Variables

Il existe plusieurs façon de définir des variables: en précisant son type ou en laissant le compilateur le « deviner ». Pour déclarer une variable de façon implicite, nous utiliserons l'opérateur `:=` :

```
var name type = value
name := value
```

Lorsque l'on utilise la manière explicite, nous ne sommes pas obligés d'attribuer une valeur à la variable. Par défaut, sa valeur sera la valeur nulle du type (`""` pour un `string` , 0 pour un `int` , etc.).

Voici quelques exemples de déclarations de variables:

```
var firstname string = "Alice"
lastname := "Dupont"
var age int
age = 20
fmt.Printf("Bonjour %s %s, tu as %d ans\n", firstname, lastname, age)
```

Une fois définie, une variable **ne peut pas changer de type**. Le bout de code suivant donnera une erreur lors de la compilation:

```
name := "Alice"
name = 1337
// error: cannot use 1337 (untyped int constant) as string value in assignment
```

[2]En réalité, les `error` sont de type `interface` .

De plus, une même variable **ne peut pas être déclarée deux fois dans le même bloc**:

```
name := "Alice"
name = 1337
// error: name redeclared in this block
```

2.3. Constantes

Les constantes sont des variables spéciales qui contiennent une valeur qui est **immutable**. Elles sont définies comme les variables, mais avec le mot clé `const`. Elles **ne peuvent pas être déclarées** avec la construction `:=`.

Les constantes peuvent être de type `byte`, `string`, `bool` ou de tout autre type pouvant contenir des **valeurs numériques**.

```
// déclarations simples
const pi = 3.14
const g = 9.81

// déclarations groupées
const (
    age = 42
    height = 180
)
```

Exercice

- Créer un programme qui déclare une variable de **manière explicite** `myNum1`, lui attribue la valeur 50 et l'affiche.
- Faire la même chose avec `myNum2`, en la déclarant de **manière implicite**.

3. Opérateurs

Il existe 4 grandes familles d'opérateurs en Go:

3.1. Opérateurs arithmétiques

Les opérateurs arithmétiques sont utilisés pour réaliser des opérations mathématiques comme l'addition, la soustraction etc.

Rôle	Opérateur	Exemple
Addition	`+`	`a + b`
Soustraction	`-`	`a - b`
Multiplication	`*`	`a * b`
Division	`/`	`a / b`
Modulo	`%`	`a % b`
Incrément	`++`	`a++`
Décrément	`--`	`a--`

3.2. Opérateurs d'assignement

Les opérateurs d'assignement sont utilisés pour assigner une valeur à une variable. Ils peuvent être **simples** ou **composés**:

Type	Opérateur	Exemple	Identique à
Simple	`=`	`var a = b`	`a := b`
Composé	`:=`	`a := b`	`bar a = b`
Composé	`+=`	`a += b`	`a = a + b`
Composé	`-=`	`a -= b`	`a = a - b`
Composé	`*=`	`a *= b`	`a = a * b`
Composé	`/=`	`a /= b`	`a = a / b`
Composé	`%=`	`a %= b`	`a = a % b`

3.3. Opérateurs relationnels

Les opérateurs relationnels sont utilisés pour **comparer deux valeurs** entre elles.

Opérateur	Exemple	Description
`==`	`a == b`	Retourne `true` si `a` est égal à `b`, `false` sinon.
`!=`	`a != b`	Retourne `true` si `a` est différent de `b`, `false` sinon.
`>`	`a > b`	Retourne `true` si `a` est supérieur à `b`, `false` sinon.
`<`	`a < b`	Retourne `true` si `a` est inférieur à `b`, `false` sinon.

Opérateur	Exemple	Description
`>=`	`a >= b`	Retourne `true` si `a` est supérieur ou égal à `b`, `false` sinon.
`<=`	`a <= b`	Retourne `true` si `a` est inférieur ou égal à `b`, `false` sinon.

3.4. Opérateurs logiques

Les opérateurs logiques sont utilisés pour réaliser des opérations logiques (AND, OR, XOR…) avec une ou plusieurs expressions.

Opérateur	Exemple	Description
`&&`	`exp1 && exp2`	Retourne `true` si les expressions `exp1` **et** `exp2` sont `true`, sinon `false`.
`||`	`exp1 || exp2`	Retourne `true` si les expressions `exp1` **ou** `exp2` sont `true`, sinon `false`.
`!`	`!exp`	Retourne `true` si l'expression `exp` est `false`, sinon `false`.

> **Note**
>
> Il existe d'autres opérateurs logiques (`<<` , `>>` , …), mais ils ne seront pas étudiés dans le cadre de ce cours.

4. Conditions

En programmation, une **instruction conditionnelle** est une instruction qui effectue différents calculs en fonction de l'évaluation d'une condition booléenne (`true` / `false`).

En Go, les principales instructions conditionnelles sont:

4.1. If/Else

La structure en if/else est très similaire à ce que l'on peut voir en C ou en Python, **sans parenthèses**:

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) != 2 {
      fmt.Println("wrong number of args")
  }

  if os.Args[1] == "Alice" {
      fmt.Println("Alice in Wonderland")
  } else if os.Args[1] == "Bob" {
      fmt.Println("Bob the Builder")
  } else {
      fmt.Println("No title with this name found")
  }
}
```

Conseil sur l'utilisation des `else` et le happy path

Il est considéré comme étant une bonne pratique de limiter le plus possible les `else`, pour garder le *happy path* le plus à gauche possible et ainsi améliorer la lisibilité. Par exemple:

```
if len(os.Args) == 1 {
    // happy path, do stuff
} else {
    // error and quit
}
```

peut devenir:

```
if len(os.Args) != 1 {
    // error and quit
}
// happy path, do stuff
```

Dans la plupart des cas, une inversion de la condition dans le `if` ainsi qu'un `return` / `break` / `continue` suffisent.

4.2. Switch

Le switch est une alternative aux if/else à répétition, permettant un code plus lisible et compacte.

```
package main

import (
  "fmt"
  "os"
)

func main() {
  if len(os.Args) != 2 {
      fmt.Println("wrong number of args")
  }

  switch os.Args[1] {
      case "Alice":
          fmt.Println("Alice in Wonderland")
      case "Bob":
          fmt.Println("Bob the Builder")
      default:
          fmt.Println("No title with this name found")
  }
}
```

Dans l'exemple ci-dessus, nous essayons de matcher les valeurs « Alice » ou « Bob » afin d'afficher une sortie personnalisée. Le mot clé `default` permet d'avoir un cas par défaut pour toutes les conditions qui ne matchent pas les autres.

> **À retenir**
>
> Attention, les cases d'un switch sont évalués **séquentiellement** ! L'ordre est donc important.

Dans un switch, il est possible de spécifier plusieurs valeurs pour un case en utilisant une virgule:

```
switch time.Now().Weekday() {
    case time.Saturday, time.Sunday:
        fmt.Println("It's the weekend")
    default:
        fmt.Println("It's a weekday")
}
```

5. Boucles

En Go, il n'existe qu'un seul mot-clé permettant de faire de boucles: `for`. Elles peuvent prendre plusieurs formes:

- boucle infinie:

```
for {
  fmt.Println("Hello ESSIR")
  time.Sleep(1 * time.Second)
}
```

- boucle avec condition unique:

```
i := 1
for i <= 3 {
  fmt.Println(i)
  i = i + 1
}
```

- boucle avec limites:

```
for i := 0; i < 10; i++ {
  fmt.Println(i)
}
```

- itération autour d'un slice ou d'un array:

```
names := []string{"Alice", "Bob", "Charlie"}
for i, name := range names {
  fmt.Printf("Hello %s, you are at position %d\n", name, i+1)
}
```

5.1. Contrôle des boucles

Il est possible de contrôler le flow d'exécution des boucles avec les mots-clés `break` et `continue`:

- `break` permet de retourner à la fonction englobante (quitter le `for`);
- `continue` saute au début du `for` pour une nouvelle itération.

```
for i := 0; i < 10; i++ {
  if i % 2 == 0 {
    continue // Retour au début de la boucle for pour une nouvelle itération
  }

  if i == 5 {
    fmt.Println("five")
    break // Quitte la boucle for
  }
  fmt.Println(i)
}
```

La sortie du programme précédent est donc:

```
1
3
five
```

Exercice

1. Écrire une boucle `for` qui a pour limite basse 0, limite haute `os.Args[1]`, un incrément de 1, et affiche:
   - « foo » si l'index est entièrement divisible par 3;
   - « bar » si l'index est entièrement divisible par 5;
   - « baz » si l'index est entièrement divisible par 3 et par 5.

   Toutes les vérifications nécessaires au bon fonctionnement du programme sont attendues.
2. Écrire un programme qui prend une chaine de caractères s en argument, et l'affiche après avoir passé tous les caractères majuscules en minuscule. Par exemple:
   - Hello ⟶ hello
   - hELLO ⟶ hello
   - hello ⟶ hello

   Seul l'usage des packages `fmt` et `os` est autorisé. `strings` & co. ne le sont pas.

6. Structures de données

6.1. Tableaux (*arrays*)

Un *array* (tableau) est une séquence numérotée d'éléments d'**un seul et même type**. Le nombre d'éléments d'un array n'est jamais négatif et est connu lors de la compilation.

La longueur du tableau peut être découverte en utilisant la fonction `len`. Les éléments peuvent être adressés par les indices entiers 0 à `len(a)-1`.

Les types de tableaux sont toujours unidimensionnels mais peuvent être composés pour former des types multidimensionnels:

```
[32]int // tableau unidimensionnel
[8][8]byte // tableau bidimensionnel
```

Pour déclarer un tableau de 10 chaînes de caractères vides nommé `arr` (par exemple), on utilise les formes:

```
var arr[10] string
```

Stocker des valeurs à différents index se fait de la manière suivante:

```
arr[0] = "Hello"
arr[1] = "ESSIR"
```

Ici, le premier index du tableau contiendra la string « Hello » et le second index du tableau contiendra la string « ESSIR ».

> **!** **À retenir**
>
> Comme dans la plupart des langages de programmation, **la première valeur d'un tableau est stockée à l'index numéro 0** !
>
> Ce sera également valable pour les slices que nous verrons plus tard.

6.2. Slices

Un slice est une séquence numérotée d'éléments d'**un seul et même type**. À la différence d'un array où la taille est explicitée à la création, le nombre d'éléments d'un slice **n'est pas connu lors de la compilation**.

Comme pour un array, La longueur d'un slice peut être découverte en utilisant la fonction `len`. Les éléments peuvent être adressés par les indices entiers 0 à `len(sl)-1`.

Les slices sont (plus ou moins) équivalent à ce que l'on peut faire avec `malloc()` et `free()` en C par exemple, c'est à dire allouer de l'espace mémoire de manière variable.

Un slice vide se déclare de la façon suivante:

```
var sl []type
// ou
sl := []type{}
```

On peut également initialiser un slice avec des valeurs:

```
sl := []string{"Abra", "cada", "bra"}
```

Pour ajouter un élément à un slice, on utilise la fonction `append()`:

```
var sl []string
fmt.Println(sl) // sortie: []
fmt.Println(len(sl)) // sortie: 0

sl = append(sl, "foo")
fmt.Println(sl) // sortie: [foo]
fmt.Println(len(sl)) // sortie: 1
```

Il existe plusieurs méthodes pour enlever un élément d'un slice, en fonction de si l'ordre des éléments a une importance ou non.

- Si l'ordre **est important**:

```
var idx int // idx est l'index de l'élément à enlever
slice = append(slice[:idx], slice[idx+1:]...)
```

- Si l'ordre **n'est pas important**:

```
var idx int // idx est l'index de l'élément à enlever
slice[idx] = slice[len(slice)-1]
slice = slice[:len(slice)-1]
```

6.3. Maps

Les maps sont un type de données qui **associent une clé à une valeur**. Elles sont souvent présentes dans les autres langages sous le nom de *hash tables*, ou `dict` (en Python).

C'est l'un des types de données les plus puissant en informatique, car permettent une recherche d'éléments rapide: en Go, l'accès aux valeurs d'une map dont les clés sont finies (`int`) a une complexité[3] de $O(1)$ (et en moyenne $O(N \log N)$ pour des `string`).

Une map ressemble à:

```
map[KeyType]ValueType
```

où `KeyType` peut être n'importe quel type comparable[4] et `ValueType` n'importe quel type (même une autre map !).

[3] https://www.bigocheatsheet.com/ °.

[4] Les types non-comparables sont ceux pour lesquels nous ne pouvons pas utiliser d'opérateurs de comparaison, par exemple les fonctions ou les maps.

Nous pouvons prendre l'exemple d'une map `m` qui lie des clés sous forme de chaîne de caractères à des entiers:

```
var m map[string]int
```

Les maps sont des types de **référence**, comme les pointeurs. Dans l'exemple ci-dessus, la valeur de `m` est `nil`, elle ne pointe pas vers une map initialisée. Une map `nil` se comporte comme une map vide lorsque l'on essaie de lire, mais essayer d'écrire dans une map non-initialisée cause un *runtime panic*.

Pour initialiser une map, il faut utiliser la fonction `make`:

```
m = make(map[string]int)
```

La fonction `make` alloue et initialise une *hash map* et retourne une valeur qui pointe vers elle.

Assigner des valeurs à des clés est assez simple:

```
m["zone"] = 51
```

Cette déclaration assigne la valeur 51 à la clé « zone ». Pour retrouver la valeur de la clé « zone », on peut faire comme suit:

```
num := m["zone"] // num sera un int de valeur 51
```

Si l'on essaie de récupérer la valeur d'une clé qui n'existe pas, on obtient la valeur zéro du type de la valeur en question:

```
num := m["foobar"]
// num sera un entier égal à 0, car la valeur zéro d'un int est 0
```

La fonction *builtin* `len` permet de connaître le nombre d'éléments que contient une map:

```
fmt.Println(len(m))
// sortie: 1
```

La fonction *builtin* `delete` permet de supprimer un élément de la map:

```
delete(m, "zone")
fmt.Println(m["zone"])
// sortie: 0
```

Il est possible d'assigner deux variables lors de la substitution d'une clé d'une map:

```
num, ok := m["zone"]
```

`num` contiendra la valeur associée à la clé « zone » (ou la valeur zéro du type de la valeur), et `ok` sera un booléen qui sera à `true` si la clé a une valeur associée, `false` sinon. Cette structure permet de faire des conditions facilement:

```
num, ok := m["zone"]
if ok {
    fmt.Printf("c'est la zone %d !\n", num)
} else {
    fmt.Println("la zone est vide :(")
}
```

Pour initialiser une map avec plusieurs valeurs, on peut utiliser la construction suivante:

```
notes := map[string]int{
    "Alice": 17,
    "Bob": 11,
    "Charlie": 14,
    "Ed": 4,
}
```

6.4. Structures

Une struct (structure) est un **type défini par le ou la développeur·euse** qui permet de regrouper/combiner des éléments de types éventuellement différents en un seul et même type. Les éléments d'une struct sont les **champs** de la structure.

Voici un exemple de définition d'une struct:

```
type person struct {
    name string
    age int
}
```

Cette structure définie un type `person` qui contient les champs `name` et `age`.

Il est possible de créer et d'interagir avec une struct comme suit:

```
p := person{} // création d'une struct vide
fmt.Printf("Hello %s, tu as %d ans\n", p.name, p.age)
// sortie: Hello tu as ans

p.name = "Alice"
p.age = 42
fmt.Printf("Hello %s, tu as %d ans\n", p.name, p.age)
// sortie: Hello Alice, tu as 42 ans

// création d'une struct avec des valeurs
p2 := person{
    name: "Bob",
    age: 69,
}
fmt.Printf("Hello %s, tu as %d ans\n", p2.name, p2.age)
// sortie: Hello Bob, tu as 69 ans
```

Exercice

1. Créer une struct pouvant contenir les informations présentes sur une carte d'identité (nom, prénom, date de naissance…).
2. Écrire un programme qui prend en argument le chemin vers un fichier contenant du texte, et qui compte le nombre de fois que chaque mot apparaît dans ce fichier.

7. Erreurs

En Go, il est **idiomatique** de communiquer les erreurs via une **valeur de retour explicite** et **distincte**. Cela contraste avec les exceptions utilisées dans des langages comme Java et Ruby et la valeur unique parfois utilisée en C.

Cette approche permet de voir facilement quelles sont les fonctions qui renvoient des erreurs et de les gérer en utilisant les mêmes constructions que celles utilisées pour toutes autre tâche.

> **!** **À retenir**
>
> Une bonne gestion des erreurs est une **condition essentielle** pour écrire des programmes robustes

Par convention, les erreurs sont la dernière valeur de retour des fonctions et sont de type `error`, qui est une interface[5] du langage.

Si vous avez déjà fait un peu de Go ou fait quelques recherches pour les exercices de ce cours, vous êtes sûrement déjà tombés sur des variables du type `error`. Prenons la fonction suivante définie dans le package `os` :

```
func ReadFile(name string) ([]byte, error)
```

Sans trop s'attarder sur la syntaxe des fonctions qui sera étudiée au prochain chapitre, cette fonction prend un argument en entrée et retourne deux valeurs, la dernière étant du type `error`. Ainsi, nous utiliserons cette fonction comme suit:

```
data, err := os.ReadFile("file.txt")
if err != nil {
        log.Fatal(err)
}
```

On peut également se créer nos propres erreurs avec la fonction `New()` du package `errors` :

```
err := errors.New("this is the error message")
```

Au-delà de la simple vérification que l'erreur est différente de `nil`, on peut également se baser sur son contenu:

[5] Par manque de temps, les interfaces ne seront malheureusement pas étudiées dans ce cours. Cependant, leur connaissance et leur maîtrise reste indispensable pour bien comprendre le langage et son fonctionnement.

```
var errDivideByZero = errors.New("divide by zero")

func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, errDivideByZero
    }
    return a / b, nil
}

func main() {
    res, err := divide(42, 0)
    if err != nil {
        if errors.Is(err, errDivideByZero) {
            fmt.Println("I know this error: it's divide by zero!")
        } else {
            fmt.Println("I do not know this error")
        }

        return
    }

    fmt.Printf("%d/%d = %d\n", 42, 0, res)
}
```

Le bien connu package `fmt` permet également de créer des erreurs, en ajoutant des données dynamiques aux messages:

```
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("can't divide '%d' by zero", a)
    }
    return a / b, nil
}
```

8. Fonctions et méthodes

8.1. Fonctions

Une fonction est un bloc de **code réutilisable** qui prend **zéro ou plus paramètres** et retourne **zéro ou plus valeurs**. Le principal objectif d'une fonction est d'**éviter la répétition de code**.

Voici la définition d'une fonction nommée `greet` :

```
func greet(name string) {
    fmt.Printf("Hello %s\n", name)
}
```

Cette fonction reçoit un paramètre (nommé `name` de type `string`) et l'affiche. Elle ne retourne aucune valeur.

On peut l'appeler ailleurs dans le code comme suit:

```
person := "Alice"
greet(person)
greet("Bob")
// sortie:
// Hello Alice
// Hello Bob
```

Voici un autre exemple de fonction qui retourne une valeur de type `bool` (un booléen): `true` si elle connaît le nom donné en paramètre, `false` sinon:

```
func isKnown(name string) bool {
    knownNames := [4]string{"Alice", "Bob", "Charlie", "Delphine"}
    for _, v := range knownNames {
        if name == v {
            return true
        }
    }
    return false
}
```

Pour une fonction qui retourne plusieurs valeurs, il faudra les mettre entre parenthèses:

```
func isKnown(name string) (bool, int) {
    knownNames := [4]string{"Alice", "Bob", "Charlie", "Delphine"}
    for i, v := range knownNames {
        if name == v {
            return true, i
        }
    }
    return false, -1
}
```

En Go, **il n'y a pas de limite** quant au nombre de paramètres qu'une fonction peut recevoir ou de valeurs qu'elle peut retourner. On peut très bien imaginer la fonction suivante:

```
func newPerson() (firstname, lastname string, age, height int, address,city
string) {
    return "Foo", "Bar", 42, 180, "Rue Haieverlor", "Lyon"
}
```

En revanche, on voit rapidement que cette façon de faire n'est pas idéale, surtout lorsque l'on doit faire évoluer son code. C'est là que les `struct` rentrent en jeu.

Nous pouvons par exemple définir la structure suivante représentant une personne:

```
type person struct {
    firstname, lastname string
    age, height int
    address, city string
}
```

Puis créer une fonction qui retourne une nouvelle instance de cette structure, avec les champs initialisés aux valeurs par défaut:

```
func newPerson() person {
    p := person{
        firstname: "Foo",
        lastname: "Bar",
        age: 42,
        height: 180,
        address: "Rue Haieverlor",
        city: "Lyon",
    }
    return p
}
```

Il est également possible de définir une fonction qui reçoit (ou retourne) un **nombre inconnu de paramètres** lors de la compilation:

```
func greet(names ...string) {
    for _, name := range names {
        fmt.Printf("Hello %s!\n", name)
    }
}
```

Dans cet exemple, `names` est de type `[]string` (un slice de string). Ces fonctions sont appelées **fonctions variadiques**.

8.2. Méthodes

Les méthodes sont un type de fonction qui permettent d'interagir avec un *receiver*. La méthode peut accéder aux propriétés du *receiver* et les modifier.

```go
type rectangle struct {
    width, height int
}

func (r rectangle) area() int {
    return r.width * r.height
}

func (r rectangle) perimeter() int {
    return 2 * r.width + 2 * r.height
}

func main() {
    r := rectangle{
        width: 10,
        height: 5,
    }

    fmt.Println(r.area()) // sortie: 50
    fmt.Println(r.perimeter()) // sortie: 30
}
```

Exercice

1. Créer une fonction `swap` qui prend en paramètres deux entiers et les retourne dans l'ordre inverse (si `a` et `b` en entrée, retourne `b` et `a` en sortie).
2. Créer une fonction `stringInSlice` qui prend en paramètres une string, un slice de string et retourne `true` si la string est dans le slice, `false` sinon.
3. Créer une fonction racine carrée: étant donné un nombre x, nous voulons trouver le nombre z pour lequel z^2 est le plus proche de x. Les ordinateurs calculent généralement la racine carrée de x à l'aide d'une boucle. En commençant par une estimation de z, nous pouvons ajuster z en fonction de la proximité de z^2 avec x, produisant une meilleure estimation, selon la formule: $z = \frac{z^2-x}{2z}$.

   Cette approche est appelée Méthode de Newton-Raphson[6].

   Un bon point de départ pour z est 1, et 10 un bon nombre de calculs. Afficher z à chaque boucle pour suivre son évolution.

[6] https://fr.wikipedia.org/wiki/M%C3%A9thode_de_Newton

9. Packages et modules

9.1. Packages

En Go, un *package* est une collection de de un ou plusieurs fichiers `.go` liés entre eux. Le principal objectif des packages est de grouper du code.

Chaque fichier source doit commencer par la définition du package auquel il appartient, avec l'instruction `package`. Dans l'exemple ci-dessous, notre fichier `main.go` appartient au package `main` :

```
package main

import (
    "fmt"
    "math/rand"
)

func main() {
    fmt.Println("Mon chiffre favori est", rand.Intn(10))
}
```

Le package `main` est un package particulier car il indique que son code est fait pour être exécuté (et pas seulement importé comme dans le cas des librairies). C'est pour cette raison que tous les packages `main` doivent contenir une fonction `main()` qui sera le point d'entrée de l'exécutable.

9.1.1. Importation d'autres packages

Par convention, le nom du package est le même que le dernier élément du chemin d'importation. Par exemple, le paquet `math/rand` comprend des fonctions qui commencent par l'indication du package `rand`.

Un module, que l'on peut apparenter à un projet, est défini avec la commande:

```
go mod init <nom du package>
```

Par convention, le nom du module est l'URL du projet. C'est pour cela que l'on trouve souvent des packages ayant pour nom quelque chose comme: `github.com/creator/project`.

Une module peut contenir plusieurs packages, dont le package `main` si il contient un exécutable.

Donner le lien du package en nom permet également d'importer facilement des packages dans son propre code. Si par exemple je veux utiliser le package `sirupsen/logrus` pour écrire des logs:

```go
package main

import (
    "math/rand"
    "github.com/sirupsen/logrus"
)

func main() {
    logrus.Info("Mon chiffre favori est", rand.Intn(10))
}
```

Les packages importés sans URL ou lien particulier sont généralement dans la librairie standard de Go. C'est le cas pour `fmt`, `os`, `ioutil`, `errors`...

Il est possible de renommer localement le nom d'un package, pour par exemple faciliter son appel dans le code. Si l'on repend l'exemple de `sirupsen/logrus` :

```go
package main

import (
    "math/rand"
    log "github.com/sirupsen/logrus"
)

func main() {
    log.Info("Mon chiffre favori est", rand.Intn(10))
}
```

9.2. Exportation

Comme on a pu le voir à nombreuses occasions, lorsque l'on appelle un fonction d'un package, le nom de la fonction commence par une majuscule. Cela signifie que la fonction est exportée: elle est accessible en dehors du package.

> **!** **À retenir**
>
> Pour exporter une **variable**, un **type**, une **constante** ou une **fonction**, il faut que son nom commence par une **majuscule**.

```go
// inaccessible par d'autres packages
var name string

// accessible par d'autres packages qui importent le notre
func Greeting(name string) {
    fmt.Printf("Hello %s\n", name)
}
```

10. Concurrence

10.1. *Concurrency is not parallelism*

Avant d'aborder les différents mécanismes que propose Go pour réaliser de la programmation **concurrente**, il est important de comprendre la différence entre la concurrence et le parallélisme.

Le calcul **parallèle** nous permet de décomposer des problèmes complexes en petits morceaux et d'exécuter ces petits problèmes **simultanément**. Cependant, la simultanéité des calculs dépend du nombre de processeurs. Si le système n'a qu'un seul processus, chaque étape du problème sera exécutée de manière séquentielle.

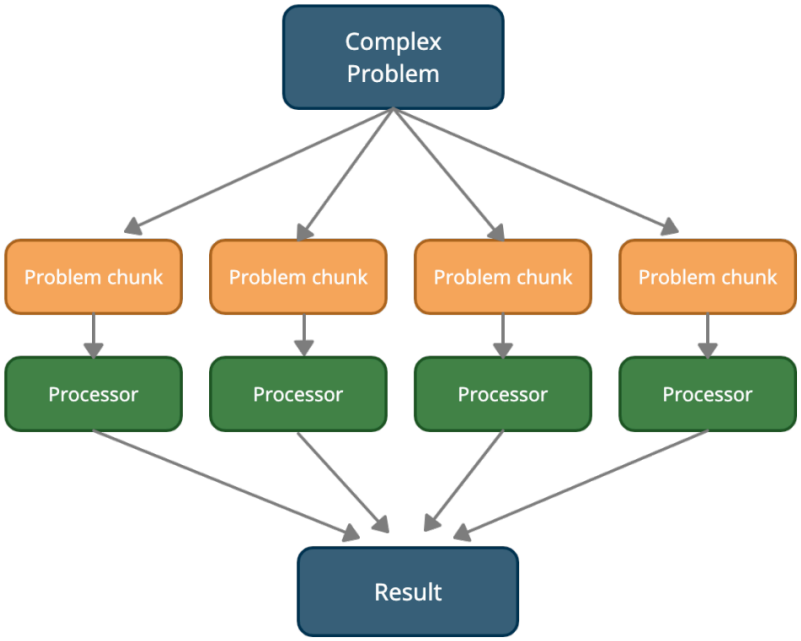

Fig. 1. – Partage de tâches d'un programme parallèle

Un programme **concurrent** peut ou ne peut pas être exécuté de manière parallèle. La concurrence consiste à structurer un programme de manière à ce que deux ou plus tâches **puissent être en cours simultanément**, tandis que le parallélisme permet d'**exécuter deux ou plusieurs tâches simultanément**. Notez que si le parallélisme nécessite plus d'un processeur ou d'un thread, la concurrence ne l'exige pas.

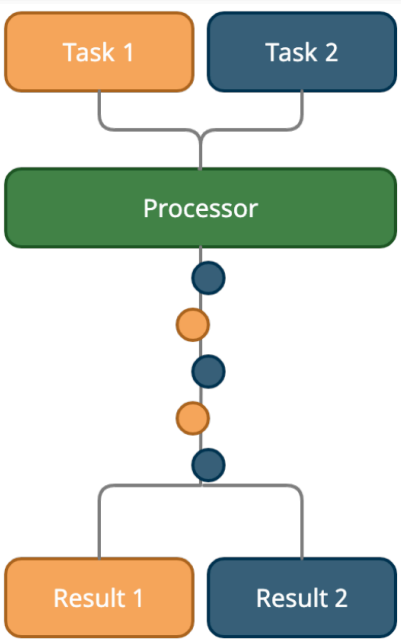

Fig. 2. – Partage de tâches d'un programme concurrent

10.2. Goroutines

Une goroutine est une fonction qui est exécutée de manière concurrente. On peut la voir comme étant un *thread* léger. N'importe quelle fonction peut être exécutée en tant que goroutine grâce au mot-clé `go` :

```
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")
}
```

Les goroutines sont un moyen extrêmement performant d'exécuter des tâches en parallèle. Il est possible d'en avoir **plusieurs centaines de milliers** en même temps sans aucun problème.

La taille de la stack minimale requise pour un goroutine est de 2KB, ce qui leur donne une très faible empreinte mémoire si bien utilisées.

> ⚠ **Avertissement**
>
> Les goroutines s'exécutent dans le **même espace d'adressage**, l'accès à la mémoire partagée doit donc être **synchronisé**.

Le package `sync` fournit des primitives utiles, mais Go propose également un moyen de synchroniser et d'envoyer des données entre goroutines: les **channels**.

10.3. Channels

Les channels sont des « tuyaux » qui permettent de relier des goroutines. Vous pouvez envoyer des valeurs dans les channels depuis une goroutine et recevoir ces valeurs dans une autre goroutine.

On peut créer un channel avec la construction:

```
myChannel := make(chan type)
```

Par exemple, pour créer un channel qui servira à envoyer des `string`:

```
messages := make(chan string)
```

Pour envoyer une valeur dans un channel, il faut utiliser la construction `channel <- value`:

```
messages := make(chan string)
messages <- "Hello World!"
```

Pour recevoir du contenu envoyé dans un channel, il faut utiliser la construction `<- channel`:

```
receivedMsg <- messages
fmt.Printf("message received: %s\n", receivedMsg)
// sortie:
// message received: Hello World!
```

> ⚠ **Avertissement**
>
> L'écriture et la lecture dans un channel sont **bloquantes** ! Si le channel est plein lorsque l'on essaie d'y écrire, ou vide lorsque l'on essaie d'y lire, l'instruction **restera en attente**.
>
> Par exemple:
>
> ```
> func main() {
>     msg := make(chan string)
>     fmt.Println(<-msg)
> }
> // fatal error: all goroutines are asleep - deadlock!
> ```

Par défaut, les channels ne sont **pas bufferisés**, ce qui veut dire qu'ils n'acceptent des écritures (`channel <-`) uniquement si il y a une lecture (`<- channel`) de prête.

Par exemple:

```go
func main() {
    msg := make(chan string)
    msg <- "Hello World!"
    fmt.Println(<-msg)
}
// fatal error: all goroutines are asleep - deadlock!
```

Les channels **bufferisés** acceptent un nombre **limité et défini** de valeurs sans forcément avoir de receveur de prêt pour ces valeurs.

Par exemple:

```go
func main() {
    msg := make(chan string, 1)
    msg <- "Hello World!"
    fmt.Println(<-msg)
}
// sortie:
// Hello, World!
```

Les channels peuvent être utilisés par des goroutines pour se synchroniser:

```go
func worker(done chan bool) {
    fmt.Println("starting work...")
    time.Sleep(5 * time.Second)
    fmt.Println("I'm done!")
    done <- true
}

func main() {
    done := make(chan bool, 1)
    go worker(done)
    <- done
}
```

Il est cependant recommandé d'utiliser les `WaitGroup` du package `sync` pour synchroniser des goroutines. Les `WaitGroup` permettent d'attendre qu'une ou plusieurs goroutines soient terminées:

```
func worker(id int) {
    fmt.Printf("worker %d starting\n", id)
    time.Sleep(5 * time.Second)
    fmt.Printf("worker %d ended\n", id)
}

func main() {
    var wg sync.WaitGroup

    for i := 0; i < 3; i++ {
        wg.Add(1)
        i := i // https://go.dev/doc/faq#closures_and_goroutines

        go func() {
            defer wg.Done()
            worker(i)
        }()
    }

    wg.Wait()
    fmt.Println("end of program")
}
```

La sortie du programme précédent sera:

```
worker 2 starting
worker 1 starting
worker 0 starting
worker 1 ended
worker 2 ended
worker 0 ended
end of program
```

Lorsque l'on utilise des channels en tant que paramètres de fonctions, il est possible de spécifier si un channel est sensé recevoir ou envoyer des valeurs:

```
func ping(pings chan<- string, msg string) {
    pings <- msg
}

func pong(pings <-chan string, pongs chan<- string) {
    msg := <-pings
    pongs <- msg
}

func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)

    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

La construction `select` du langage permet d'attendre plusieurs goroutines simultanément:

```
c1 := make(chan string)
c2 := make(chan string)

go func() {
    time.Sleep(1 * time.Second)
    c1 <- "one"
}()

go func() {
    time.Sleep(2 * time.Second)
    c2 <- "two"
}()

select {
    case msg1 := <-c1:
        fmt.Println("received", msg1)
    case msg2 := <-c2:
        fmt.Println("received", msg2)
}
```

11. Licence

Index des figures