Commentaires, variables, littéraux et affectation

Mettre des commentaires dans les programmes

- Comme leur nom l'indique, les commentaires servent à *commenter* votre code. Ils permettent d'en faciliter la relecture (par vous, plus tard, ou par ceux qui feront évoluer votre code — ou le corrigeront…). Typiquement, on devrait pouvoir retrouver votre algorithme dans les commentaires

- Il y a deux types de commentaires en Go :

  - Un commentaire multi-lignes est compris entre **/*** et ***/**

  - Un commentaire mono-ligne est introduit par **//** et se termine à la fin de la ligne

- Les commentaires doivent être judicieux !!!

L'opérateur de déclaration/initialisation

- Vous aurez remarqué que, pour déclarer et initialiser sa variable de boucle, **for** utilise un nouvel opérateur :

```
for cpteur := 1; cpteur <= 10; cpteur++ {…}
```

- En réalité, l'opérateur **:=** peut s'utiliser n'importe où lorsqu'on veut déclarer une variable en lui donnant une valeur initiale.

- Au lieu d'écrire `var cpteurCotes int = 1`, on peut écrire `cpteurCotes := 1`. En ce cas, Go déduit de la valeur (1, ici) le type de la variable (un **int**, ici). C'est la syntaxe préférée des Gophers. Mais vous êtes libres de préférer la première version (sauf dans le **for**, où la forme **:=** est obligatoire...)

Utilisation de l'opérateur de déclaration/initialisation

Sans...

```
func main() {
    var (
        seuil, cpteur, pourcentage, seuil int
        hauteur                           float64
    )

    hauteur = 100.98
    pourcentage = 10
    seuil = 50

    cpteur = 0
    for int(hauteur) > seuil {
        hauteur = nouvelleHauteur(hauteur, pourcentage)
        fmt.Printf("%.2f\n", hauteur)
        cpteur += 1
    }

    fmt.Printf("Le seuil de %v a été atteint en %v rebonds\n", seuil, cpteur)
}
```

Avec...

```
func main() {
    hauteur := 100.98
    pourcentage := 10
    seuil := 50

    cpteur := 0
    for int(hauteur) > seuil {
        hauteur = nouvelleHauteur(hauteur, pourcentage)
        fmt.Printf("%.2f\n", hauteur)
        cpteur += 1
    }

    fmt.Printf("Le seuil de %v a été atteint en %v rebonds\n", seuil, cpteur)
}
```

Compléments sur l'affectation en Go

- Comme la plupart des langages, Go dispose de l'**affectation combinée aux opérateurs** qui permet d'économiser la frappe tout en restant lisible :

```
age := 20
age += 1                    // équivalent à : age = age + 1

prix := 100
prix *= 1 + tva/100         // équivalent à : prix = prix * (1 + tva/100)

ch := "bonjour"
ch += " les amis"           // équivalent à : ch = ch + " les amis"
```

- Cette écriture est possible avec tous les opérateurs binaires : **+**, **-**, *****, **/** et **%**

Compléments sur l'affectation en Go

- On peut également effectuer des **affectations en parallèle** :

```
val1, val2 = 10, 100          // équivalent à : val1 = 10 ; val2 = 100

val1, val2 = val2, val1       // échange le contenu de val1 et val2
```

- Pour bien comprendre la deuxième ligne, rappelez-vous qu'une affectation évalue d'abord sa partie droite, **puis** affecte le résultat à sa partie gauche.

- Enfin, Go dispose des opérateurs *d'incrémentation* et de *décrémentation* : **cpteur++** augmente de 1 la valeur de cpteur, **cpteur--** la diminue de 1.

Types et système de typage

Types et système de typage

- En algorithmique et en programmation, tout objet (constante, variable ou littéral) a un **type**.

- C'est ce type qui détermine les **valeurs** que peut recevoir une variable et c'est ce type qui définit les **opérations** que l'on pourra faire avec cet objet. C'est également lui qui détermine ce que renvoie une fonction.

- Nous avons déjà rencontré le type entier (**int**), le type flottant (**float64**), le type booléen (**bool**) et le type chaîne de caractères (**string**).

- Il existe aussi des variantes permettant de préciser les tailles des représentations des entiers et des flottants, mais nous ne les présenterons pas dans ce cours.

Types et système de typage : compatibilité et conversions de types

- En Go, les types numériques ne sont **pas** compatibles entre eux : vous ne pouvez pas mélanger dans une même opération un **int** avec un **float64**.

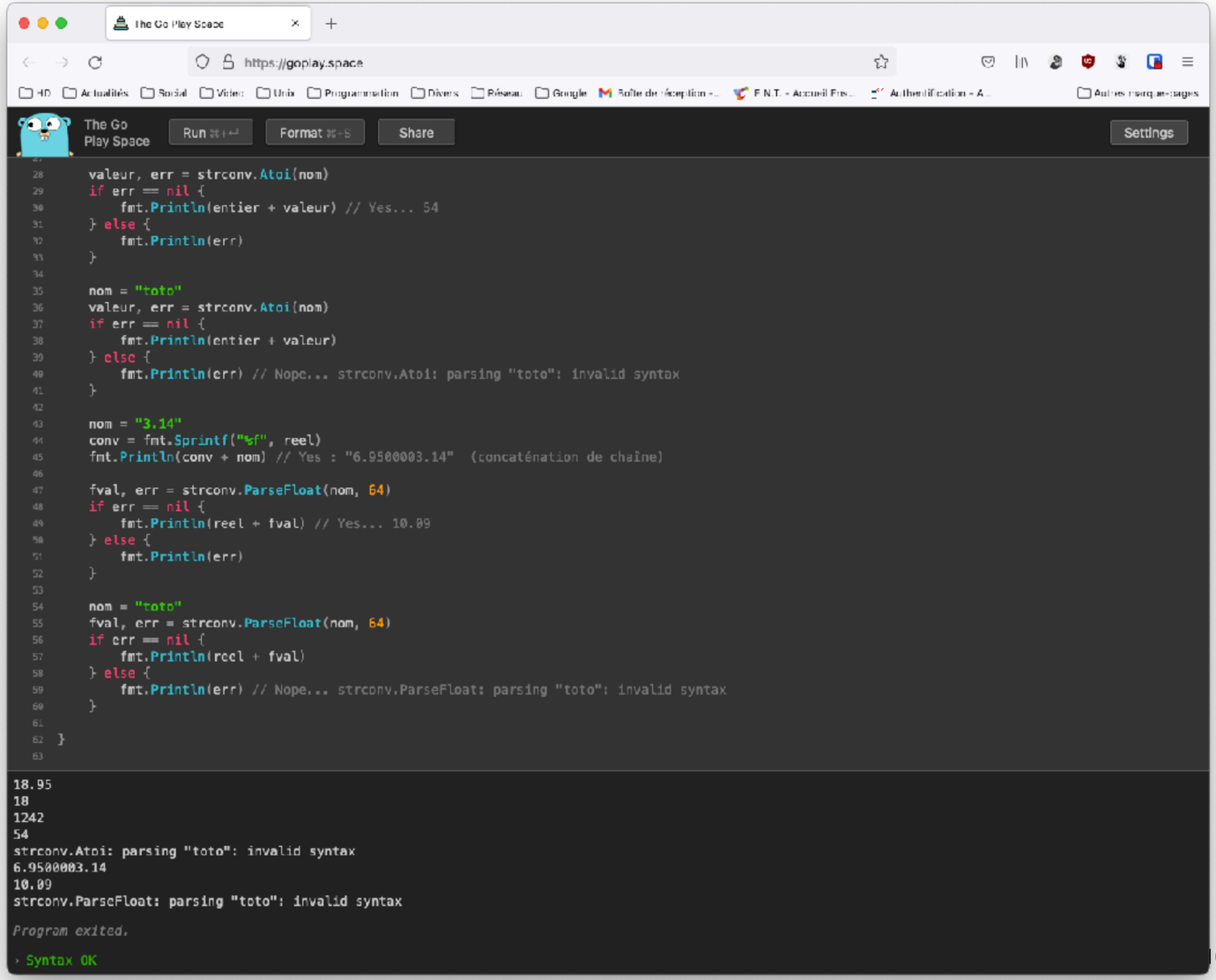
- De même, une variable de type **int** ne peut pas recevoir une valeur de type **float64** et réciproquement. Si nécessaire vous devrez convertir l'une dans le type de l'autre.

- Les **littéraux numériques *entiers*** (*42*, *-16*, par exemple) sont spéciaux : ils sont d'un type numérique "*universel*" (compatible avec tous les nombres). Ainsi, *42* peut se mélanger sans problème avec un **int** ou un **float64**.

- Évidemment, les nombres ne sont pas compatibles avec les chaînes de caractères et réciproquement. Là aussi, on dispose de fonctions permetttant de convertir l'un dans le type de l'autre lorsque cela a du sens.

Types et système de typage : compatibilité et conversions de types

[Cliquez sur ce lien]



Conversions entre nombres : résumé

- Pour convertir un nombre *nbre* en **int** : **int(*nbre*)**
- Pour convertir un nombre *nbre* en **float64** : **float64**(*nbre*)
- Pour convertir un nombre *nbre* en **string** : **fmt.Sprint(*nbre*)**
- Pour convertir un **string** en **int** : **nbre, err = strconv.Atoi(*chaine*)**
- Pour convertir un **string** en **float64** : **nbre, err = strconv.ParseFloat(*chaine*, 64)**

Gestion des erreurs : ignorer les erreurs avec une variable muette

- Pour les deux dernières conversions, la variable **err** est de type **error.**

- En Go, *toute variable déclarée doit être utilisée* : ceci signifie que si vous écrivez **nbre, err = strconv.Atoi(…)**, vous **devrez** ensuite utiliser **err** pour satisfaire le compilateur…

- Si vous préférez ignorer l'erreur, vous devez alors utiliser une "variable muette" à la place de **err**, c'est à dire une variable dont le seul but est d'occuper une place dans le code : vous ne pouvez pas vous contenter d'écrire **nbre = strconv.Atoi(…)** car le compilateur se plaindra qu'il attend deux variables et que vous n'en avez donné qu'une seule (*assignment mismatch: 1 variable but strconv.Atoi returns 2 values*)

- Vous devrez donc écrire **nbre, _ = strconv.Atoi(chaine)**… où le blanc souligné représente la variable muette)

Gestion des erreurs : le type error

- Le type **error** est un nouveau type servant, comme son nom l'indique, à gérer les éventuelles erreurs.

- Une valeur **error** est initialisée par un appel à **errors.New("message d'erreur")** ou **fmt.Errorf("message d'erreur")** ou en recevant le résultat d'une fonction comme **strconv.Atoi**.

- Une valeur **error** non initialisée vaut **nil** par défaut, qui est une valeur spéciale signifiant "pas d'erreur" (on ne peut pas affecter directement nil à une variable **error**).

- Voir cet exemple.

Gestion des erreurs : le type error

- Le principe général de la gestion des erreurs en Go est :

  - que les fonctions susceptibles de rencontrer des erreurs (fonctions de conversion, fonctions de manipulation des fichiers, etc.) renvoient un résultat supplémentaire, de type **error** (voir, par exemple, **strconv.Atoi** ou **os.Open**). Ce résultat vaudra la valeur spéciale **nil** s'il n'y a pas eu d'erreur, une valeur différente sinon (nous verrons plus tard comment écrire de telles fonctions).

  - que le code qui utilise ces fonctions teste cette erreur dans un **if** ou un **for**.

Gestion des erreurs : exemples

- Attention lorsqu'on ne teste pas les erreurs renvoyées par les conversions de chaînes vers des nombres !

- Si l'on tente de convertir la chaîne "toto" en entier ou en flottant, par exemple, le nombre obtenu sera fixé à **zéro** et vous ne saurez pas que c'est faux… Voir ce lien.

- La bonne façon de faire consiste à tester l'erreur, qui doit valoir **nil** si tout s'est bien passé. Voir ce lien.

- D'autres langages, comme Python et Java, utilisent le mécanisme des *exceptions* pour traiter ce type d'erreurs (ce qui est un peu plus lourd à gérer).

Les expressions

Les expressions

- Une **expression** est une combinaison de valeurs (constantes, variables et littéraux) et d'opérations (opérateurs ou appels de fonctions) qui *renvoie une valeur*.
  - Par exemple, **2 * 3.14 * rayon** est une expression.
- Autrement dit, une ***expression est tout ce qui peut produire une valeur***.
- Comme toute valeur, une expression est donc "typée" : on parle d'*expression entière*, d'*expression booléenne*, d'*expression flottante*, d'*expression chaine de caractères*. L'expression de l'exemple précédent est une **expression flottante** (ou réelle) à cause de la présence de 3.14 (et *rayon* doit donc également être un flottant, sinon, le compilateur se plaindra...).

Les expressions : exemples

- Soit le code suivant :

```
age := 30
taille := 1.88
obstacle := true
nom := "Martin"
```

- **age**, **taille**, **obstacle** et **nom** sont des *variables* respectivement entière, flottante, booléenne et chaîne. Ce sont aussi des expressions (simples).
- **30**, **1.88**, **true** et **"Martin"** sont des *littéraux* respectivement entier, flottant, booléen et chaîne. Ce sont aussi des expressions (simples).
- **age + 2** est une expression entière, **taille < 2.00** est une composée booléenne, **age / 2** est une expression entière, **len(nom)** est une expression entière, **len(nom) + 1 < len("Dupont")** est une expression booléenne, **taille / 2** est une expression flottante, etc. Toutes ces expressions sont composées (non simples).

Sous-expressions

- Une expression composée est formée de plusieurs **sous-expressions** qui peuvent, ou non, être délimitées par des parenthèses (la présence de parenthèses ajoute à la lisibilité et permet de préciser l'ordre d'évaluation de l'expression) :

  - **(2 * longueur) + (4 * largeur)** est une expression numérique formée de deux sous-expressions (qui elles-mêmes sont composées). Ici, les parenthèses ont un sens : la valeur de cette expression est différente de l'expression **2 * (longueur + 4) * largeur**…

  - **age >= 10 && age <= 100** est une expression booléenne formée de deux sous-expressions composées : ici, les parenthèses ne sont pas obligatoires car l'opérateur **&&** est moins prioritaire que les opérateurs de comparaisons : ceux-ci sont donc appliqués avant...

- Une sous-expression est elle-même une expression et permet donc de décomposer un calcul en plusieurs étapes.

Sous-expressions et priorités des opérateurs

- Comme en arithmétique et comme dans tous les langages, certains opérateurs sont **prioritaires** par rapport à d'autres.

- Pour les opérateurs arithmétiques, par exemple, les opérateurs de multiplication et de division sont prioritaires par rapport aux opérateurs d'addition et de soustraction : l'expression **3 + 4 * 2** sera donc évaluée comme **3 + (4 * 2)** puisque la multiplication s'effectue avant l'addition… le résultat sera différent de **(3 + 4) * 2**…

- Pour lever les doutes et éviter les ambiguïtés, il est fortement conseillé de parenthéser les sous-expressions afin de mieux représenter l'ordre des opérations…

Les nombres et leurs opérateurs

Les nombres et leurs opérateurs

- Le type entier (**int**) permet de stocker et de manipuler les nombres entiers, **positifs ou négatifs**. La taille d'un **int** dépend de la plateforme : sur les ordinateurs actuels, il est codé sur 64 bits (ce qui permet de représenter des entiers allant de -9223372036854775808 à 9223372036854775807). On peut précéder un littéral entier d'un **+** ou d'un **-**.

- Le type flottant (**float64**) permet de stocker et manipuler les nombres réels **positifs ou négatifs** (de -1.8 x 10^{308} à 1.8 x 10^{308}). La « virgule » est représentée par un point. Un flottant peut aussi s'exprimer sous forme "scientifique".

```
num := 12                  // int
num := -78                 // int

num := 3.14                // float64
num := -0.12               // float64
num := .12                 // float64
num := 3.                  // float64
num := 1e100               // float64
num := -2.34e-10           // float64
```

Attention : ce code n'est pas correct
num ne peut pas changer de type !

Les nombres et leurs opérateurs : autres opérations mathématiques

- Outre les opérateurs arithmétiques déjà vus (**+**, **-**, *****, **/** et **%**), Go fournit également plusieurs fonctions mathématiques : **Abs(x)**, **Min(x, y)**, **Max(x, y)** qui prennent toujours des opérandes **float64** et renvoient un résultat de type **float64.**

- Le paquetage **math** (qu'il faut importer avec **import "math"**), fournit également un grand nombre d'autres fonctions et plusieurs constantes.

- Exemple d'utilisation de la constante **math.Pi** et de la fonction **math.Pow** :

```
fmt.Println(math.Pi * math.Pow(3, 2))          // 28.274333882308138
x, y := 3, 2
fmt.Println(math.Pi * math.Pow(x, y))          // Erreur compilation ! Pourquoi ?
```

Les nombres et leurs opérateurs : autres opérations mathématiques

Pour plus d'informations sur les fonctions et constantes du paquetage **math**, servez-vous de la commande **go doc** :

Default

```
[Dalek ~]
 └─ % go doc math
package math // import "math"

Package math provides basic constants and mathematical functions.

This package does not guarantee bit-identical results across architectures.

const E = 2.71828182845904523536028747135266249775724709369995957496696763 ...
const MaxFloat32 = 3.40282346638528859811704183484516925440e+38 ...
const MaxInt8 = 1<<7 - 1 ...
func Abs(x float64) float64
func Acos(x float64) float64
func Acosh(x float64) float64
func Asin(x float64) float64
func Asinh(x float64) float64
func Atan(x float64) float64
func Atan2(y, x float64) float64
func Atanh(x float64) float64
func Cbrt(x float64) float64
func Ceil(x float64) float64
func Copysign(x, y float64) float64
func Cos(x float64) float64
func Cosh(x float64) float64
func Dim(x, y float64) float64
func Erf(x float64) float64
func Erfc(x float64) float64
func Erfcinv(x float64) float64
func Erfinv(x float64) float64
func Exp(x float64) float64
func Exp2(x float64) float64
func Expm1(x float64) float64
func FMA(x, y, z float64) float64
func Float32bits(f float32) uint32
func Float32frombits(b uint32) float32
func Float64bits(f float64) uint64
func Float64frombits(b uint64) float64
func Floor(x float64) float64
func Frexp(f float64) (frac float64, exp int)
```

```
[Dalek ~]
 └─ % go doc math.Abs
package math // import "math"

func Abs(x float64) float64
    Abs returns the absolute value of x.

    Special cases are:

        Abs(±Inf) = +Inf
        Abs(NaN) = NaN

[Dalek ~]
 └─ %
```

Lire la documentation locale : go doc et godoc

- La commande **go doc *paquetage*** permet d'obtenir la liste des principales constantes et des fonctions définies dans le paquetage indiqué : par exemple, **go doc fmt** ou **go doc math**.

- L'option **-all** demande d'afficher toutes les constantes et les descriptions de chaque fonction : par exemple, **go doc -all math**

- Pour afficher la description détaillée d'une fonction particulière, utilisez la commande **go doc *paquetage.fonction***. Par exemple **go doc fmt.Println**

- Pour afficher la description des types ou des fonctions prédéfinies du langage, faites **go doc builtin** (et **go doc builtin.int** ou **go doc builtin.append**, par exemple)

- La commande **godoc -http=localhost:6060** lance un serveur web sur le port 6060 de votre machine à partir duquel vous pouvez consultez toute la documentation en faisant pointer votre navigateur vers l'URL **localhost:6060** (vous pouvez choisir un autre port supérieur à 1024...).