11X001 – Introduction à la programmation des algorithmes

3.1 Variables et Mutabilité

François Fleuret

https://fleuret.org/francois

19 Octobre, 2020

Le contenu de ce document a été en grande partie repris du cours de Jean-Luc Falcone.

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Variables

Notion de variables

Dans tout ce que nous avons vu jusque là, nous avons uniquement utilisé des expressions fonctionnelles.

Nous n'avons pas vu de moyens de modifier des entités persistantes.

Notion de variables

Dans tout ce que nous avons vu jusque là, nous avons uniquement utilisé des expressions fonctionnelles.

Nous n'avons pas vu de moyens de modifier des entités persistantes.

Ce concept est pourtant central à l'approche historique de la programmation des ordinateurs, et correspond plus directement à la réalité physique de la mémoire.

Notion de variables (cont.)

Dans la plupart des langages, les valeurs sont manipulées avec des variables.

En Python par exemple les constantes n'existent pas. On peut indiquer à partir de Python 3.8 que certaines valeurs ne doivent pas changer.

Notion de variables (cont.)

En C ou en Java les quantités définies sont des variables par défaut et il faut spécifier explicitement quand ce n'est pas le cas.

```
int main(int argc, char **argv) {
  int x = 3;
  x = 4;
  const int y = 4;
  y = 5;
  exit(EXIT_SUCCESS);
}
```

```
./a.c: In function ‘main’:
./a.c:18:5: error: assignment of read-only variable ‘y’
   y = 5;
```

Notion de variables (cont.)

```
class Main {
  public static void main(String[] args) {
    int x = 3;
    x = 4;
    final int y = 4;
    y = 5;
  }
}

Main.java:6: error: cannot assign a value to final variable y
    y = 5;
    ^
1 error
compiler exit status 1
```

Variables en Swift

Swift offre des variables qui permettent de représenter de telles quantités modifiables.

On peut définir une variable avec le mot clé `var`.

La syntaxe est similaire à celle de `let` pour les constantes.

```
var i: UInt8 = 0
var s: String = "12"
var x: Bool = true
```

Inférence du type

La valeur de la déclaration permet d'inférer le type de la variable:

```
var i = 0    // Int
var s = "12" // String
var x = true // Bool
```

Variables non initialisées (1)

Une variable peut être déclarée, c'est à dire que l'on peut indiquer son identifiant et son type, avant d'être initialisée, c'est à dire avant de lui donner une première valeur.

```
var x: UInt8  // Déclaration
print("Hello, world !")
x = 122       // Initialisation
```

Il est obligatoire dans ce cas d'annoter le type de la variable lors de la déclaration, puisque l'inférence de type n'est pas possible sans valeur.

Variables non initialisées (2)

Une variable doit être initialisée, c'est à dire que l'on doit lui donner une première valeur, avant d'être utilisée:

```
var x: Int
print(x) // error: variable 'x' used before
         // being initialized
x = 12
```

Alors que:

```
var x: Int
x = 12
print(x)  // Affiche 12
```

Assignation (1)

Une variable peut être modifiée par assignation:

```
var x = 5 // Déclaration et initialisation
print(x)  // Affiche 5

x = 12    // Assignation
print(x)  // Affiche 12

x = -2    // Assignation
print(x)  // Affiche -2
```

Assignation (2)

Une assignation doit respecter le type de la variable:

```
var x = 5 // Infère un Int
x = 12.5  // error: cannot assign value of
          // type 'Double' to type 'Int'
```

Utiliser plutôt:

```
var x: Double = 5

x = 12.5
```

Exemple

L'ordre d'exécution est beaucoup plus important avec des variables!

```
var x = 12
var y = x
y *= 2
print(x)  // Affiche 12
print(y)  // Affiche 24
```

Exemples (cont.)

```
var x = 12
var y = x
x += 1
y *= 2
print(x)
print(y)
```

Exemples (cont.)

```
var x = 12
var y = x
x += 1
y *= 2
print(x)
print(y)
```

affiche

```
13
24
```

Exemples (cont.)

```
var x = 12
x += 1
var y = x
y *= 2
print(x)
print(y)
```

Exemples (cont.)

```
var x = 12
x += 1
var y = x
y *= 2
print(x)
print(y)
```

affiche

```
13
26
```

Variables constantes

Si le compilateur détecte qu'une variable n'est jamais modifiée, il suggère d'utiliser une constante à la place:

```
func chose(x: Int) -> (Int, Int) {
  var a = 5
  var b = 5
  a = a + x
  return (a, b)
}

a.swift:3:7: warning: variable 'b' was never mutated; consider
changing to 'let' constant
  var b = 5
  ~~~ ^
  let
```

Exemple: Calcul de la norme d'un vecteur

```
struct Vector3D {
  let x, y, z: Double
}
```

Version purement fonctionnelle:

```
func norm( v: Vector3D ) -> Double {
  let sq = {x: Double in x*x}
  let sum_sq = sq(v.x) + sq(v.y) + sq(v.z)
  return sqrt(sum_sq)
}
```

Exemple: Calcul de la norme d'un vecteur (cont.)

```
struct Vector3D {
  let x, y, z: Double
}
```

Avec une variable:

```
func norm( v: Vector3D ) -> Double {
  var n = v.x * v.x
  n = n + v.y * v.y
  n = n + v.z * v.z
  n = sqrt(n)
  return n
}
```

Opérateurs de réassignation

On peut abréger la syntaxe d'une réassignation dans le cas où c'est une modification simple de la valeur courante:

```
var n = 1
n += 1      // n = n + 1
n *= 10     // n = n * 10
n -= 5      // n = n - 5
n /= 2      // n = n / 2
print(n)    // ???
```

Opérateurs de réassignation (cont.)

```
func norm( v: Vector3D ) -> Double {
  var n = v.x * v.x
  n += v.y * v.y
  n += v.z * v.z
  n = sqrt(n)
  return n
}
```

Modèle d'évaluation

Modèle d'évaluation

En programmation fonctionnelle on a utilisé un modèle d'évaluation par substitution.

En programmation impérative, on doit utiliser un modèle d'évaluation basé sur des environnements d'exécution.

Environnements d'exécution

Un environnement d'exécution est un répertoire de variables ou de constantes et de leurs valeurs associées:

- L'environnement est vide au début du programme,
- chaque déclaration de variables ou de constante ajoute une entrée,
- chaque assignation modifie une entrée.

Modèle d'évaluation

Pour évaluer/analyser un programme impératif il faut:

- l'exécuter ligne par ligne en suivant l'ordre d'exécution,
- évaluer immédiatement et complétement les expressions rencontrées, et
- mettre à jour l'environnement d'exécution.

Environnements d'exécution: Exemple

```
print("Bonjour")  // { }
var x = 3         // { x = 3 }
var y = 5         // { x = 3, y = 5 }
x = 4             // { x = 4, y = 5 }
var z = 11        // { x = 4, y = 5, z = 11 }
y = 3             // { x = 4, y = 3, z = 11 }
var w = x + y     // { x = 4, y = 3, z = 11, w = 7 }
z = 1             // { x = 4, y = 3, z = 1, w = 7 }
print(x, y, z, w) // { x = 4, y = 3, z = 1, w = 7 }
```

affiche

```
Bonjour
4 3 1 7
```

Environnements d'exécution: Exemple

```
var x: Int      // { x=? }
var y = 2.5     // { x=?, y=2.5 }
x = 3           // { x=3, y=2.5 }
y += Double(x) // { x=3, y=5.5 }
let z = 2*x     // { x=3, y=5.5, z=6 }
```

Environnements d'exécution: Exemple

Le même code sans variable:

```
let x = 3
let y0 = 2.5
let y1 = y0 + Double(x)
let z = 2*x
```

Ordre d'exécution (1)

```
var x = 3     // { x=3 }
var y = x + 1 // { x=3, y=4 }
x += 2        // { x=5, y=4 }
var z = x     // { x=5, y=4, z=5 }
z *= y        // { x=5, y=4, z=20 }
```

Ordre d'exécution (2)

```
var x = 3      // { x=3 }
x += 2         // { x=5 }
var y = x + 1  // { x=5, y=6 }
var z = x      // { x=5, y=6, z=5 }
z *= y         // { x=5, y=6, z=30 }
```

Ordre d'exécution (3)

```
var x = 3     // { x=3 }
var z = x     // { x=3, z=3 }
x += 2        // { x=5, z=3 }
var y = x + 1 // { x=5, y=6 z=3 }
z *= y        // { x=5, y=6, z=18 }
```

Substitution (1)

```
let x0 = 3
let y = x0 + 1
let x1 = x0 + 2
let z0 = x1
let z1 = z0 * y
```

Substitution (1)

```
let x0 = 3
let y = x0 + 1
let x1 = x0 + 2
let z0 = x1
let z1 = z0 * y
```

```
let z1 = z0 * y
let z1 = x1 * y
let z1 = (x0 + 2) * y
let z1 = (3 + 2) * y
let z1 = (3 + 2) * (x0 + 1)
let z1 = (3 + 2) * (3 + 1)
let z1 = (3 + 2) * (3 + 1)
let z1 = 20
```

Substitution (2)

```
let x0 = 3
let x1 = x0 + 2
let z0 = x1
let y = x0 + 1
let z1 = z0 * y
```

Substitution (2)

```
let x0 = 3
let x1 = x0 + 2
let z0 = x1
let y = x0 + 1
let z1 = z0 * y
```

```
let z1 = z0 * y
let z1 = x1 * y
let z1 = (x0 + 2) * y
let z1 = (3 + 2) * y
let z1 = (3 + 2) * (x0 + 1)
let z1 = (3 + 2) * (3 + 1)
let z1 = (3 + 2) * (3 + 1)
let z1 = 20
```

Fonctions et procédures

Procédure

On appelle procédure une fonction qui ne retourne pas de valeur.

Il ne peut donc pas s'agir d'une fonction pure, et son seul intérêt de produire des effets de bords:

- modification de variables,
- entrée / sorties,
- interactions avec le système,
- etc.

Une procédure en Swift n'a pas de type de retour.

Exemple de procédures

```
var x = 10

func increment() {
  x += 1
}

func reset() {
  x = 0
}

increment()
increment()
print(x)

reset()
increment()
increment()
print(x)
```

affiche

```
12
2
```

Exemple de procédures (cont.)

```
var x = 10              // { x = 10 }

func increment() {
  x += 1
}

func reset() {
  x = 0
}

increment()             // { x = 11 }
increment()             // { x = 12 }
print(x)                // { x = 12 }

reset()                 // { x = 0 }
increment()             // { x = 1 }
increment()             // { x = 2 }
print(x)                // { x = 2 }
```

Les arguments des fonctions ne sont pas mutables

```
func truc( _ x: Int ) -> Int {
   var y = 1
   x = x * 2
   y += x
   return y
}

print( truc(2) )

error: cannot assign to value: 'x' is a 'let' constant
   x = x * 2
   ^
```

Variables et fonctions

```
func truc( _ x: Int ) -> Int {
   var y = 1
   let z = x * 2
   y += z
   return y
}
```

Ordre d'évaluation des arguments

Ordre d'évaluation des arguments

```
var x = 1

func increments() -> Int {
  x+=1
  return x
}

func halves() -> Int {
  x /= 2
  return x
}
```

Ordre d'évaluation des arguments

```
var x = 1

func increments() -> Int {
  x+=1
  return x
}

func halves() -> Int {
  x /= 2
  return x
}

func compute(a: Int, b: Int) -> (Int, Int) {
  return (a, b)
}

let c = compute(a: increments(), b: halves())

print(x, c)
```

Ordre d'évaluation des arguments

```
var x = 1

func increments() -> Int {
  x+=1
  return x
}

func halves() -> Int {
  x /= 2
  return x
}

func compute(a: Int, b: Int) -> (Int, Int) {
  return (a, b)
}

let c = compute(a: increments(), b: halves())

print(x, c)
```

affiche

```
1 (2, 1)
```

Ordre d'évaluation des arguments (cont.)

Comprendre ce que fait

```
let c = compute( a: increments(), b: halves() )
```

demande une analyse détaillée.

- En Swift les arguments des fonctions sont évalués de gauche à droite.
- L'ordre d'évaluation diffèrent selon les langages et cela peut-être très ambiguë.
- Si une fonction n'est pas pure, il vaut mieux l'appeler en dehors de toute autre expression.

```
let x1 = increments()
let x2 = halves()
let c = compute( a: x1, b: x2 )
```

Factorielle: implémentation récursive fonctionnelle

```
func factorial( _ n: UInt ) -> UInt {

  func iter( _ i: UInt, _ prod: UInt ) -> UInt {
    if( i > n ) {
      return prod
    } else {
      return iter( i+1, prod*i )
    }
  }

  return iter( 1, 1 )
}
```

Factorielle: implémentation récursive impérative et fausse

```
func factorial( _ n: UInt ) -> UInt {
  var i: UInt = 1
  var prod: UInt = 1

  func iter() -> UInt {
    if( i > n ) {
      return prod
    } else {
      i += 1
      prod *= i
      return iter()
    }
  }

  return iter()
}
```

Factorielle: implémentation récursive impérative et correcte

```
func factorial( _ n: UInt ) -> UInt {
  var i: UInt = 1
  var prod: UInt = 1

  func iter() -> UInt {
    if( i > n ) {
      return prod
    } else {
      prod *= i
      i += 1
      return iter()
    }
  }

  return iter()
}
```

Exemples / exercices

Exemple / exercice (1)

```
var x: Int
var y: Int
let z = 5
y = z
x = y
y = y + 3
x = x + 4
print(x)
print(y)
```

Exemple / exercice (1)

```
var x: Int
var y: Int
let z = 5
y = z
x = y
y = y + 3
x = x + 4
print(x)
print(y)
```

affiche

```
9
8
```

Exemple / exercice (2)

```
import Foundation

func blah(q: Double) -> Double {
  let q = Double.pi
  return cos(q)
}

var r = 0.0

let z = blah(q: r)

print(r, z)
```

Exemple / exercice (2)

```
import Foundation

func blah(q: Double) -> Double {
  let q = Double.pi
  return cos(q)
}

var r = 0.0

let z = blah(q: r)

print(r, z)
```

affiche

```
0.0 -1.0
```

Exemple / exercice (3)

```
var r = 0.0
let s = 1.0
var t = 3.0

s += t
s += r
print(r, s, t)
```

Exemple / exercice (3)

```
var r = 0.0
let s = 1.0
var t = 3.0

s += t
s += r
print(r, s, t)
```

affiche

```
var3_exo.swift:6:3: error: left side of mutating operator isn't mutable: 's' is a 'let'
s += t
~ ^
var3_exo.swift:3:1: note: change 'let' to 'var' to make it mutable
let s = 1.0
^~~
var
var3_exo.swift:7:3: error: left side of mutating operator isn't mutable: 's' is a 'let'
s += r
~ ^
var3_exo.swift:3:1: note: change 'let' to 'var' to make it mutable
let s = 1.0
^~~
var
```

Exemple / exercice (4)

```
var r = 0.0
var s = 1.0
var t = 3.0

t += s
t += r
print(r, s, t)

r += t
r += s
print(r, s, t)

s += t
s += r
print(r, s, t)
```

Exemple / exercice (4)

```
var r = 0.0
var s = 1.0
var t = 3.0

t += s
t += r
print(r, s, t)

r += t
r += s
print(r, s, t)

s += t
s += r
print(r, s, t)
```

affiche

```
0.0 1.0 4.0
5.0 1.0 4.0
5.0 10.0 4.0
```

Exemple / exercice (5)

```
var x = 15

func doit(i: Int) -> Int {
  x = i
  return x
}

print(x)
print(doit(i: 3))
print(x)
```

Exemple / exercice (5)

```
var x = 15

func doit(i: Int) -> Int {
  x = i
  return x
}

print(x)
print(doit(i: 3))
print(x)
```

affiche

```
15
3
3
```

Exemple / exercice (6)

```
var x = 15

func doit(i: Int) -> Int {
  x = x * 2
  if i > 0 {
    return doit(i: i - 1)
  } else {
    return x
  }
}

print(x)
print(doit(i: 3))
print(x)
```

Exemple / exercice (6)

```
var x = 15

func doit(i: Int) -> Int {
  x = x * 2
  if i > 0 {
    return doit(i: i - 1)
  } else {
    return x
  }
}

print(x)
print(doit(i: 3))
print(x)
```

affiche

```
15
240
240
```

FIN