11X001 – Introduction à la programmation des algorithmes

4 Récapitulation

François Fleuret

https://fleuret.org/francois

14 Décembre, 2020

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

UNIVERSITÉ
DE GENÈVE

FACULTY OF SCIENCE

Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.

Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.
- Ces bits sont organisés par groupes de 8 (octets, ou *bytes*).
- Un octet peut être interprété de plusieurs manières, en particulier comme une écriture en base 2 d'une valeur entière entre 0 et 255.

Dans la mémoire d'un ordinateur:

- Toutes les données sont une suite de zéros et de uns (bits).
- Ce choix de représentation de l'information découle de la facilité de représenter des valeurs oui/non, présence/absence dans un objet physique.
- Ces bits sont organisés par groupes de 8 (octets, ou *bytes*).
- Un octet peut être interprété de plusieurs manières, en particulier comme une écriture en base 2 d'une valeur entière entre 0 et 255.

00000000	0	00000001	1
00000001	1	00000010	2
00000010	2	00000011	3
00000100	4	00000100	4
00001000	8	00000101	5
00010000	16	00000110	6
00100000	32	00000111	7
01000000	64	10000001	129
10000000	128	11111111	255

Types (cont.)

Dans tous les langages:

- Toutes les données ont un type associé.
- Le type définit et restreint les opérations possibles sur les données.

Au niveau des langages on distingue:

- Typage statique vs. typage dynamique.
- Typage fort vs. typage faible.

Typage statique

- Un type est associé à chaque expression du programme avant son exécution.
- Propre des programmes compilés.
- Les types doivent être déclarés ou inférés.
- Par exemple: Java, C, Swift, Pascal, Haskell, Typescript.

Typage dynamique

- Un type est associé à chaque expression du programme pendant son exécution.
- Propre des programmes interprétés.
- Pas de déclaration de types.
- Par exemple: Python, Scheme, Javascript, Bash.

Typage fort

- Les conversions d'un type à un autre doivent être explicites.
- Par exemple: Python, Haskell, Swift, OCaml.

Typage faible

- Le type d'une expression peut changer implicitement durant l'exécution.
- Par exemple: Java, C, Perl, PHP.

Typage Statique et Fort

Plus rigide:

- force à prendre les décisions avant,
- plus difficile de changer après (sauf bonne encapsulation).

Plus sûr:

- possibilité d'analyse et de raisonnement,
- vérification statique à la compilation,
- permet d'exprimer des contraintes au niveaux des types.

Types primitif en Swift

Plusieurs types de **nombres entiers**:

`Int8`, `UInt8`, `Int16`, ..., `Int64`, `UInt64`.

Les **nombres décimaux** sont représentés par les types `Float` et `Double`.

Les **valeurs logiques** par le type `Bool` qui peut prendre les valeurs `true` et `false`.

Types primitif en Swift (cont.)

Le type `String` représente du texte arbitraire

- Pas de limitation de taille (mémoire)
- Tous les caractéres unicodes: Chiffres, alphabet latin, cyrillique, idéogramme chinois, emojis, ...

Le type `Character` représente un caractère dans une `String`.

Expressions

Une expression est une combinaison d'éléments de syntaxe qui peut être évaluée en un résultat.

Une déclaration est une combinaison d'éléments de syntaxe qui ne produit pas de résultat mais qui peut:

- Modifier le contexte de compilation: types, constantes, fonctions.
- Modifier le contexte d'exécution: contrôle du flux.
- Produire un effet de bord: affectations, entrées/sorties.

Constantes

On déclare une **constante** en associant un identifiant à une expression à l'aide du mot réservé `let`. Son type peut être **inféré** ou explicitement spécifié.

Une assignation est une déclaration. Elle ne retourne pas de valeur.

Constantes

On déclare une **constante** en associant un identifiant à une expression à l'aide du mot réservé `let`. Son type peut être **inféré** ou explicitement spécifié.

Une assignation est une déclaration. Elle ne retourne pas de valeur.

```
let g = 9.8065
let three = 2 + 1
let trois = three
let weight = g * 25.0
let debugMode = true
```

Opérateurs

Les expressions combinent des opérateurs.

Opérateurs arithmétiques binaires:

- Addition `+`
- Soustraction `-`
- Multiplication `*`
- Division `/`
- Reste de division `%` (seulement entre entiers)

Opérateurs arithmétiques unaire:

- Opposé `-` (change le signe d'une expression)

Opérateurs (cont.)

Opérateurs booléens binaires:

- Conjonction (et) `&&`
- Disjonction (ou) `||`

Opérateurs booléen unaire:

- Négation (non) `!`

Opérateurs (cont.)

On peut comparer des valeurs numériques à l'aide des **opérateurs de comparaisons** qui retournent des `Bool`.

- `==`: égal
- `>`: plus grand
- `>=`: plus grand ou égal
- `<` : plus petit
- `<=` : plus petit ou égal

Attention

Les comparaisons d'expressions de types numériques décimaux doivent être faites avec prudence.

Attention

Les comparaisons d'expressions de types numériques décimaux doivent être faites avec prudence.

```
  1> let x = 0.3
x: Double = 0.29999999999999999
  2> let y = 0.2 + 0.1
y: Double = 0.30000000000000004
  3> print(x == y)
false
```

Tableaux

- Représente une liste de valeurs du même type.
- On accède à une valeur d'un tableau en utilisant son indice.
- La première valeur a l'indice 0, la dernière valeur a l'indice $n - 1$ où n est la taille du tableau.
- La propriété `count` permet de connaître la taille d'un tableau.

```
  1> let t = [1, 2, 3]
t: [Int] = 3 values {
  [0] = 1
  [1] = 2
  [2] = 3
}
  2> print( t.count )
3
  3> print( t[0] + t[2] )
4
  4> print( t[5] )
Fatal error: Index out of range
```

Valeurs optionnelles

- Représente un valeur possiblement indéfinie.
- On note `T?` où `T` est le type de la valeur qui peut être absente.
- Par exemple `Bool?`, `String?`.
- On crée un optionnel:
  - soit en passant la valeur si elle existe,
  - soit en passant `nil` si elle n'existe pas.

```
let from: UInt? = 12
let to: UInt? = nil

struct Rectangle {
  let height: Double
  let width: Double
  let fillColor: Color?
}
let r = Rectangle(height:12, width:5, fillColor:nil)
```

Un optionnel est “emballé” (wrapped), il faut donc le “déballer” (unwrap) avant de l'utiliser.

- On peut forcer le déballage avec un point d'exclamation.
- Cela produira une erreur d'exécution si l'optionnel n'est pas défini.

```
let from: UInt? = 12
let to: UInt? = nil

print( from + 2 )  // Erreur de compilation
print( from! + 2 ) // Affiche 14
print( to! + 2 )   // Erreur d'exécution
```

On peut également déballer une valeur optionnelle avec l'opérateur binaire `??` qui permet de passer une valeur par défaut :

```
print( (from ?? 100) + 2 ) // Affiche 14
print( (to ?? 100)  + 2 ) // Affiche 102
```

Structure

- Possède des propriétés, chacun avec un type et un identifiant.
- On crée une instance d'une structure en fournissant la valeur de chaque propriété. Il faut indiquer le nom de chaque propriété et respecter l'ordre de définition.
- On accède aux propriétés d'une structure avec l'opérateur point `.`

```
struct Item {
  let name: String
  let available: Bool
}

let x = Item( name: "iPhone 21", available: true )

print( x.name )   // Affiche "iPhone 21''
print( !x.available ) // Affiche false
```

Types dérivés (cont.)

Tuples

- Défini uniquement par l'ordre des types qui le composent.
- Par exemple

```
(UInt8, UInt8, UInt8)
(String, Boolean)
(Int)
(Int, (Double, Double))
()
```

Énumération

- Type ayant un nombre fini de valeurs.
- Par exemple

```
enum TrafficLight {
  case green
  case yellow
  case red
}
```

Ces différentes valeurs peuvent être paramétrées par un tuple

```
enum ExtendedInt {
  case minusInf, plusInf
  case value(Int)
}
```

Fonctions pures

En programmation en général, une fonction est juste un sous-programme.

Le concept de **fonction pure** en programmation fonctionnelle est très similaire à ce même concept en mathématiques. Une telle fonction

- accepte des arguments (entrée) et retourne une valeur (sortie),
- retourne toujours la même valeur pour des arguments donnés,
- elle ne modifie pas ses arguments, ou l'état du programme,
- elle n'accède pas à d'autres valeurs que ses arguments,
- elle n'a pas d'"effets de bords" (affichage, etc.)

Fonctions pures (cont.)

```
func add( a: Int, b: Int ) -> Int {
  return a + b
}

let q = add( a: 2, b: 12 )

print(q) // Affiche 14
```

Chaque fonction a un type, formé du type de ses arguments et du type de la valeur retournée. Par exemple `add` ci-dessus a le type `(Int, Int) -> Int`.

Portée lexicale

La portée lexicale d'un identifiant (fonctions, constante, variable) est la portion du programme dans laquelle cet identifiant est défini

- Les déclarations au niveau supérieur ont une portée globale.
- Les déclarations entre accolades ont une portée locale.
- En cas de redéfinition, la portée la plus locale l'emporte.
- Les arguments d'une fonctions ont une portée locale à celle-ci.

Portée lexicale (cont.)

```
let a = 100  // Global
let b = 20   // Global
let c = 3    // Global

func truc( x: Int ) -> Int { // truc globale, mais x local
  let b = -x    // b local, cache b global
  return b * c
}

let w = truc(x: a) // w global
```

Portée lexicale (cont.)

On peut définir des fonctions dans une fonctions, celles-ci sont locales à cette dernière.

Par exemple:

```
func f( x: Double, y: Double ) -> Double {
  func sq( a: Double ) -> Double {
    return a * a
  }
  return sq(a: x) + sq(a: y)
}
```

Surcharge de fonctions

On peut définir plusieurs fonctions avec le même nom si elles différent par

- le nombre ou types des arguments, ou
- les noms d'appel des arguments, ou
- le type de retour.

Swift pourra déterminer selon le contexte de l'appel de quelle fonction il s'agit.

If. . . Else. . .

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
if x >= 0 {
  return x
} else {
  return -x
}
```

If. . . Else. . .

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
if x >= 0 {      Condition
  return x
} else {
  return -x
}
```

If. . . Else. . .

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
if x >= 0 {
  return x
} else {
  return -x
}
```

Condition

Cas vrai

If. . . Else. . .

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
if x >= 0 {        Condition
  return x         Cas vrai
} else {
  return -x        Cas faux
}
```

If. . . Else. . .

La déclaration `if...else...` permet de choisir entre deux chemins d'exécution possibles:

```
if x >= 0 {        Condition
  return x         Cas vrai
} else {
  return -x        Cas faux
}
```

Exemple:

```
func inverse( _ x: Int ) -> Double? {
  if x == 0 {
    return nil
  } else {
    return 1 / Double(x)
  }
}
```

Switch. . . Case. . .

La construction `switch / case` permet de moduler le déroulement du programme selon la valeur d'un `enum`.

```
enum ExtendedInt {
  case minusInf, plusInf
  case value(Int)
}

func add(x: ExtendedInt, k: Int) -> ExtendedInt {
  switch x {
  case ExtendedInt.minusInf:
    return ExtendedInt.minusInf
  case ExtendedInt.plusInf:
    return ExtendedInt.plusInf
  case ExtendedInt.value(let v):
    return ExtendedInt.value(v + k)
  }
}
```

Évaluation par substitution

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
3 > 13
```

Évaluation par substitution

Si on se limite aux éléments vus, on peut évaluer toute expression par substitutions successives.

Par exemple:

```
1 + 2 * 3 - 4 > 2 * 7 - 1
1 + 6 - 4 > 2 * 7 - 1
1 + 6 - 4 > 14 - 1
7 - 4 > 14 - 1
3 > 14 - 1
3 > 13
false
```

Évaluation par substitution (cont.)

Pour évaluer une fonction par substitutions:

1. On substitue l'appel de la fonction par son corps.
2. On remplace chaque occurence de chaque argument par l'expression passée.
3. On poursuit les substitutions jusqu'à parvenir à un `return`.
4. On remplace `{ return x }` par `x`.

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
0 + 1
```

Évaluation par substitution (cont.)

```
func f( x: Int, y: Int ) -> Int {
  return x * y - x
}
```

```
f( 2+2, 1 ) + 1
f( 4, 1 ) + 1
{ return 4 * 1 - 4 } + 1
{ return 4 - 4 } + 1
{ return 0 } + 1
0 + 1
1
```

Fonctions récursives

Une fonction récursive s'appelle elle même. En général, elle contient une condition de terminaison. Lorsque celle-ci est vraie, la récursion se termine en retournant une valeur.

```
func fact( _ n: UInt64 ) -> UInt64 {
  if n <= 1 { // Condition de terminaison
    return 1
  } else {
    return n * fact( n - 1 )
  }
}
```

Fonctions récursives (cont.)

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}

print(sum([ 3, 4, 2, -7 ]))
```

L'espace d'exécution augmente à chaque appel récursif car l'ordinateur doit garder des informations en mémoire pour finir le calcul dans lequel la valeur retournée intervient.

Une fonction récursive terminale évite ce problème en ne faisant aucune opération après l'appel récursif.

Donc, par exemple, dans la définition d'une fonction `f`, on peut avoir `return f(g(x))` mais pas `return g(f(x))`.

L'espace d'exécution augmente à chaque appel récursif car l'ordinateur doit garder des informations en mémoire pour finir le calcul dans lequel la valeur retournée intervient.

Une fonction récursive terminale évite ce problème en ne faisant aucune opération après l'appel récursif.

Donc, par exemple, dans la définition d'une fonction `f`, on peut avoir `return f(g(x))` mais pas `return g(f(x))`.

Par exemple:

```
func fact( _ n: UInt64 ) -> UInt64 {
  func factRec( _ n: UInt64, _ m: UInt64 ) -> UInt64 {
    if n <= 1 {
      return m
    } else {
      return factRec( n - 1, m * n )
    }
  }
  return factRec(n, 1)
}
```

Fonctions récursives terminales (cont.)

Récursive:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}
```

Fonctions récursives terminales (cont.)

Récursive:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt) -> Int {
    if i >= x.count {
      return 0
    } else {
      return x[i] + sumFrom(i + 1)
    }
  }

  return sumFrom(0)
}
```

Récursive terminale:

```
func sum( _ x: [Int]) -> Int {
  func sumFrom( _ i: UInt, _ partialSum: Int) -> Int {
    if i >= x.count {
      return partialSum
    } else {
      return sumFrom(i + 1, x[i] + partialSum)
    }
  }
  return sumFrom(0, 0)
}
```

Fonctions d'ordre supérieur et anonymes

Une fonction est une valeur, on peut donc la manipuler en tant que telle:

```
func add(_ x: Int, _ y: Int ) -> Int {
  return x + y
}

let plus = add

print( plus( 2, 3 ) )
```

En particulier une fonction peut recevoir des fonctions comme arguments et renvoyer une fonction comme résultat.

On parle alors de fonctions d'ordre supérieur.

Fonctions d'ordre supérieur (cont.)

```
func adder(n: Int) -> (Int) -> Int {
  func add(x: Int) -> Int {
    return n + x
  }
  return add
}

let f = adder(n: 5) // f est de type (Int) -> Int

print(f(2)) // Affiche 7
```

Fonctions d'ordre supérieur (cont.)

```
func iter(f: @escaping (Int) -> Int, n: Int) -> (Int) -> Int {

  func nfRec(x: Int, k: Int) -> Int {
    if k == 0 {
         return x
    } else {
      return nfRec(x: f(x), k: k-1)
    }
  }

  func nf(x: Int) -> Int {
    return nfRec(x: x, k: n)
  }

  return nf
}

print(iter(f: adder(n: 5), n: 10)(4)) // Affiche 54
```

Fonctions anonymes

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

```
{ (i: Int) -> Bool in return i != 0 }
```

Fonctions anonymes

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

Arguments

```
{ (i: Int) -> Bool in return i != 0 }
```

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

Arguments Résultat

```
{ (i: Int) -> Bool in return i != 0 }
```

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

Arguments Résultat Corps

```
{ (i: Int) -> Bool in return i != 0 }
```

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

Arguments Résultat Corps

```
{ (i: Int) -> Bool in return i != 0 }
```

Par exemple, au lieu de:

```
func isZero( n: Int ) -> Bool {
  return n == 0
}

let n = count( x, isZero )
```

On peut définir une fonction de manière anonyme, c'est à dire sans lui associer un identifiant.

Arguments Résultat Corps

```
{ (i: Int) -> Bool in return i != 0 }
```

Par exemple, au lieu de:

```
func isZero( n: Int ) -> Bool {
  return n == 0
}

let n = count( x, isZero )
```

On peut écrire directement:

```
count( x, { (i: Int) -> Bool in return i != 0 })
```

Fonctions anonymes (cont.)

Le corps d'une fonction anonyme peut être complexe.

```
func apply(_ a: Int, _ b: Int, _ f: (Int, Int) -> Double) -> Double {
  return f(a, b)
}

print(apply(3, 4, { (a: Int, b: Int) -> Double in
  let c = abs(b - a)
  if a < c {
    return Double(c - a) / 2
  } else {
    return Double(a + b) / 2
  }
}))
```

Types génériques

Types génériques

Les types paramétriques sont définis à la compilation et permettent de généraliser une implémentation, en la rendant indépendante du type finalement utilisé

En Swift on introduit un type générique en utilisant des chevrons (<>) entre le nom de la fonction et la liste d'arguments.

Un type générique fonctionne comme un argument pour les types, qui ne change pas durant l'appel de la fonction.

On peut passer autant de types génériques que l'on veut.

Types génériques (cont.)

```
func getThree<T>( _ x: [T] ) -> (T, T, T) {
  return (x[0], x[x.count/2], x[x.count-1])
}
```

Types génériques (cont.)

```
func getThree<T>( _ x: [T] ) -> (T, T, T) {
  return (x[0], x[x.count/2], x[x.count-1])
}


print(getThree([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]))

print(getThree([ 1.0, 0.5, 0.25, 0.125, 0.0625 ]))

print(getThree([ "Prof", "Atchoum", "Dormeur", "Grincheux",
                 "Joyeux", "Timide", "Simplet" ]))
```

affiche

```
(1, 6, 11)
(1.0, 0.25, 0.0625)
("Prof", "Grincheux", "Simplet")
```

Opérations collectives

Opérations collectives

De nombreuses opérations sur les collections (*e.g.* tableaux) peuvent être décrites comme:

1. des transformations qui appliquent une même opération à chaque élément,
2. des filtres qui sélectionnent un sous-ensemble d'éléments, ou
3. des réductions qui calculent un résultat unique à partir de tous les éléments.

Opérations collectives, fonction `map`

La fonction `map` permet de transformer un tableau en appliquant une fonction sur chaque élément.

Le tableau d'origine n'est pas modifié.

Opérations collectives, fonction `map`

La fonction `map` permet de transformer un tableau en appliquant une fonction sur chaque élément.

Le tableau d'origine n'est pas modifié.

Par exemple:

```
let xs = [ 1, 2, 3 ]

let ys = xs.map( { x in x * 2 } )

print(ys)
```

affiche

```
[2, 4, 6]
```

Opérations collectives, fonction `filter`

La fonction `filter` permet de garder uniquement les éléments qui satisfont un prédicat.

Le tableau d'origine n'est pas modifié

Opérations collectives, fonction `filter`

La fonction `filter` permet de garder uniquement les éléments qui satisfont un prédicat.

Le tableau d'origine n'est pas modifié

Par exemple:

```
let words = ["machin", "chose", "truc"]

let ns = words.filter( { w in w.count < 6 } )

print(ns)
```

affiche

```
["chose", "truc"]
```

Étant données une valeur de travail initiale

$$y_0 \in \mathcal{Y}$$

et une fonction

$$f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Y},$$

la réduction appliquée au tableau de valeurs

$$x_n \in \mathcal{X},\ n = 1, \ldots, N$$

calcule la suite de valeurs de travail

$$y_n = f(x_n, y_{n-1}),\ n = 1, \ldots, N$$

et retourne y_N.

Opérations collectives, fonction `reduce` (cont.)

La méthode `reduce` implémente cette opération, et prend comme arguments y_0 et f. Elle ne modifie pas le tableau auquel elle est appliquée.

Opérations collectives, fonction `reduce` (cont.)

La méthode `reduce` implémente cette opération, et prend comme arguments y_0 et f. Elle ne modifie pas le tableau auquel elle est appliquée.

Par exemple:

```
let ns = [1, 2, 3, 4, 5]
let n = ns.reduce( 0, { y, x in x + y } )
print(n)
```

affiche

```
15
```

Opérations collectives (cont.)

Composer ces trois opérations permet de programmer des opérations complexes de manière concise et robuste.

Exemple

```
let x = [ -2, 3, 5, -11, -1 ]

print(
  x.map({ x in abs(x) })
    .filter({ x in x <= 3 })
    .reduce(0, { (y, x) in x + y } )
)
```

Variables et procédures

Variables

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

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

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

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.

Variables (cont.)

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
```

Une structure peut contenir des propriétés mutables. Il suffit de les déclarer avec `var` au lieu de `let`:

```
struct User {
  let name: String
  var password: String
}

var alice = User(name: "Alice", password: "")

alice.password = "58Ul2dF" // Change le mot de passe

let bob = User(name: "Bob", password: "")

bob.password = "4Fg001e"  // Erreur de compilation
```

Mutabilité (cont.)

De manière similaire, on peut modifier les éléments d'une variable tableau ou tuple. *E.g*

```
var xs = [1, 2, 3]
xs[0] = 10
xs[2] += 3
print(xs) // Affiche [10, 2, 6]

var zs = (1, 3.0, "Blah")
zs.1 += 4
print(zs) // Affiche (1, 7.0, "Blah")
```

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 est 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.

Boucles

Boucles

- Les boucles sont une des constructions principales de la programmation impérative.
- L'utilisation d'une boucle permet de répéter un fragment de code.
- Une condition contrôle le nombre de répétitions.
- En général, pour que cette répétition soit utile, il faut que l'état courant puisse changer d'une itération à l'autre.

Boucles `while`

- La boucle la plus "classique" est la boucle `while`
- Elle répète un bloc de déclarations, tant qu'une expression booléenne est vraie (la condition)
- Syntaxe:

```
while CONDITION {
  DECLARATION 1
  DECLARATION 2
  DECLARATION 3
  ...
}
```

Boucles `while` (cont.)

```
var a = 1

while a <= 1000 {
  print(a)
  a = a * 2
}
```

Boucles `while` (cont.)

```
var a = 1

while a <= 1000 {
  print(a)
  a = a * 2
}
```

affiche

```
1
2
4
8
16
32
64
128
256
512
```

Boucles `while` (cont.)

On peut comparer sur un exemple l'utilisation d'une boucle et de fonctions d'ordre supérieur.

```
func grandTotal( items: [Item] ) -> UInt {
  var i = 0
  var result = 0
  while i < items.count {
    result += items[i].price * items[i].amount
    i += 1
  }
  return result
}

func grandTotal2( items: [Item]) -> UInt {
  return items.map( {it in it.price*it.amount} )
              .reduce( 0, {i, j in i+j} )
}
```

Boucles `repeat ... while`

Une boucle `repeat ... while` assure au moins une exécution du bloc de déclarations.

```
repeat {
  DECLARATION 1
  DECLARATION 2
  DECLARATION 3
  ...
} while CONDITION
```

Boucles `repeat ... while`

Exemples:

```
var a = 10
while a < 3 { // Jamais exécutée
  print(a)
  a = a + 1
}

var b = 10
repeat {      // Exécutée une fois
  print(b)
  b = b + 1
} while b < 3
```

Boucles `for ... in`

Swift permet de construire directement une boucle sur un tableau (ou plus généralement une collection) avec le mot-clé `in`.

```
for IDENTIFIANT in COLLECTION {
  DECLARATION_1
  DECLARATION_2
  ...
}
```

Par exemple:

```
let days = [ "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi" ]
for day in days {
    print(day)
}
```

Sémantique par référence

Sémantique par référence

Bien que par défaut les arguments en Swift sont passés par valeur et sont traités comme des constantes, on peut explicitement spécifier qu'un argument est passé par référence.

Dans ce cas la variable utilisée au moment de l'appel de la fonction et l'argument dans la fonction font référence à la même quantité, et toute modification du second modifie la première.

Pour indiquer qu'un argument est passé par référence:

1. Un tel argument doit être annoté avec `inout` dans la définition de la fonction.
2. La variable passée doit être préfixée par `&` quand la fonction est appelée.

On ne peut bien sûr passer ni une constante ni une expression littérale.

Sémantique par référence (cont.)

```
func incr( a: inout Int ) {
  a += 1
}

var x = 14

print(x)
incr(a: &x)
print(x)
```

affiche

```
14
15
```

FIN