11X001 – Introduction à la programmation des algorithmes

2.4a Fonctions d'Ordres Supérieurs

François Fleuret

https://fleuret.org/francois

5 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

Remarque sur les fonctions récursives

Suite de Fibonacci

La suite de Fibonacci est une suite d'entiers définie de manière récursive:

$$F_0 = 0$$
$$F_1 = 1$$
$$\forall n \geq 2,\ F_n = F_{n-2} + F_{n-1}$$

Suite de Fibonacci

Une implémentation directe aurait la forme suivante.

```
func fibo1( _ n: Int ) -> Int {
  if n == 0 {
    return 0
  } else if n == 1 {
    return 1
  } else {
    return fibo1(n - 2) + fibo1(n - 1)
  }
}

print(fibo1(0), fibo1(1), fibo1(2), fibo1(3), fibo2(4), "...", fibo1(15))

0 1 1 2 3 ... 610
```

Suite de Fibonacci

Une implémentation directe aurait la forme suivante.

```
func fibo1( _ n: Int ) -> Int {
  if n == 0 {
    return 0
  } else if n == 1 {
    return 1
  } else {
    return fibo1(n - 2) + fibo1(n - 1)
  }
}

print(fibo1(0), fibo1(1), fibo1(2), fibo1(3), fibo2(4), "...", fibo1(15))

0 1 1 2 3 ... 610
```

Cette version n'est pas satisfaisante: chaque évaluation de `fibo1` génère deux appels récursifs! Le nombre total d'opérations se comporte lui même comme une suite de type Fibonacci.

Suite de Fibonacci (cont.)

Version avec un seul appel récursif, en utilisant une fonction récursive qui retourne un tuple.

```
func fibo2( _ n: Int) -> Int {
  // n -> (F_n, F_{n+1})
  func fiboRec( _ n: Int ) -> (Int, Int) {
    if n == 0 {
      return (0, 1)
    } else {
      let a = fiboRec(n - 1)
      return (a.1, a.0 + a.1)
    }
  }

  return fiboRec(n).0
}
```

Suite de Fibonacci (cont.)

Version récursive terminale.

```
func fibo3( _ n: Int) -> Int {
  // (k, F_n, F_{n+1}) -> F_{n+k}
  func fiboRec( _ k: Int, _ a: Int, _ b: Int ) -> Int {
    if k == 0 {
      return a
    } else {
      return fiboRec(k - 1, b, a + b)
    }
  }

  return fiboRec(n, 0, 1)
}
```

Fonctions comme valeur

Fonction comme valeur

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 ) )
```

Fonction comme valeur

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 ) )
```

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

Fonction comme valeur

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 ) )
```

```
plus( 2, 3 )
add( 2, 3 )
```

Fonction comme valeur

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 ) )
```

```
plus( 2, 3 )
add( 2, 3 )
{ return 2 + 3 }
```

Fonction comme valeur

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 ) )
```

```
plus( 2, 3 )
add( 2, 3 )
{ return 2 + 3 }
2+3
```

Fonction comme valeur

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 ) )
```

```
plus( 2, 3 )
add( 2, 3 )
{ return 2 + 3 }
2+3
5
```

Type de fonction

Comme on l'a vu, une fonction a un type qui est composé des types de ses arguments et de son type de retour.

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

let plus: (Int, Int) -> Int = add
```

Type de fonction

Swift vérifie le typage comme pour tout autre valeur.

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

let plus: (Double, Double) -> Double = add

./test.swift:5:38: error: cannot convert value of type '(Int, Int) -> Int'
to specified type '(Double, Double) -> Double'
```

Type `Void`

On peut définir des fonctions qui ne retournent pas de valeur et ne font que des effets de bords (*e.g.* afficher quelque chose). Le type de retour est alors `()`, ou de manière équivalente `Void`.

```
  1> func printBlah() {
  2.   print("blah")
  3. }
  4.
  5. printBlah
$R0: () -> () =
  6> printBlah()
blah
```

Tableaux de fonctions (1)

Les fonctions étant des valeurs à part entière, on peut en particulier faire des tableaux de fonctions. *E.g.*

```
func increment( _ x: Int ) -> Int {
   return x + 1
}

func decrement( y: Int ) -> Int {
   return y - 1
}

func timesTwo( x: Int ) -> Int {
  return x * 2
}

let truc = [ increment, increment, decrement, timesTwo ]
```

Tableaux de fonctions (2)

Attention: Elles doivent toutes être du même type.

```
func increment( _ x: Int ) -> Int {
   return x + 1
}

func decrement( y: Int ) -> Int {
   return y - 1
}

func timesTwo( x: Int ) -> Int {
  return x * 2
}

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

let truc = [ increment, increment, decrement, timesTwo, add ]
```

```
./test.swift:18:12: error: heterogeneous collection literal could only
be inferred to '[Any]'; add explicit type annotation if this is intentional
```

Fonctions comme propriétés

Et des fonctions peuvent apparaître comme propriétés d'une structure.

```
import Foundation

struct BoundedMapping {
  let valueMin, valueMax: Double
  let mapping: (Double) -> Double
}

func sigmoid(x: Double) -> Double {
  return 1 / (1 + exp(-x))
}

let mySigmoid = BoundedMapping(
  valueMin: 0,
  valueMax: 1,
  mapping: sigmoid
)

print(mySigmoid.mapping(4.0))
```

Fonctions d'ordre supérieur

Fonctions comme valeurs

On peut passer n'importe quelle valeur à une fonction, donc on peut passer une fonction à une fonction.

Fonctions comme valeurs

On peut passer n'importe quelle valeur à une fonction, donc on peut passer une fonction à une fonction.

Une fonction qui prend une fonction en argument est une fonction d'ordre supérieur.

Fonction d'ordre supérieur: Exemple 1

Par exemple

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x * 2
}

print( applyTwice( x: 10, f: timesTwo ) )
```

Fonction d'ordre supérieur: Exemple 1

Par exemple

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x * 2
}

print( applyTwice( x: 10, f: timesTwo ) )
```

Affiche `40.0`.

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
timesTwo( { return 20 } )
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
timesTwo( { return 20 } )
timesTwo( 20 )
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
timesTwo( { return 20 } )
timesTwo( 20 )
{ return 20*2 }
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
timesTwo( { return 20 } )
timesTwo( 20 )
{ return 20*2 }
{ return 40 }
```

Fonction d'ordre supérieur: Évaluation

```
func applyTwice( x: Double, f: (Double) -> Double ) -> Double {
  return f( f( x ) )
}

func timesTwo( x: Double ) -> Double {
  return x*2
}
```

```
applyTwice( x:  10, f:  timesTwo )
{ return timesTwo( timesTwo( 10 ) ) }
timesTwo( timesTwo( 10 ) )
timesTwo( { return 10*2 } )
timesTwo( { return 20 } )
timesTwo( 20 )
{ return 20*2 }
{ return 40 }
40
```

Fonction d'ordre supérieur et récursive (1)

Une fonction qui applique plusieurs fois une autre fonction:

```
func applyNTimes(_ x: Double, _ f: (Double) -> Double, _ n: Int) -> Double {
  if n == 0 {
    return x
  } else {
    return applyNTimes(f(x), f, n - 1)
  }
}
```

Fonction d'ordre supérieur et récursive (1)

Une fonction qui applique plusieurs fois une autre fonction:

```
func applyNTimes(_ x: Double, _ f: (Double) -> Double, _ n: Int) -> Double {
  if n == 0 {
    return x
  } else {
    return applyNTimes(f(x), f, n - 1)
  }
}

func increment(_ x: Double) -> Double {
  return x + 1
}

print(applyNTimes(11.0, increment, 5))
```

Fonction d'ordre supérieur et récursive (1)

Une fonction qui applique plusieurs fois une autre fonction:

```
func applyNTimes(_ x: Double, _ f: (Double) -> Double, _ n: Int) -> Double {
  if n == 0 {
    return x
  } else {
    return applyNTimes(f(x), f, n - 1)
  }
}

func increment(_ x: Double) -> Double {
  return x + 1
}

print(applyNTimes(11.0, increment, 5))
```

Affiche `16.0`.

Fonction d'ordre supérieur et récursive (2)

Une fonction qui applique une série de fonctions:

```
func applyAll(_ x: Double, _ f: [ (Double) -> Double ]) -> Double {
  func applyFrom(_ x: Double, _ n: Int) -> Double {
    if n >= f.count {
      return x
    } else {
      return applyFrom(f[n](x), n + 1)
    }
  }

  return applyFrom(x, 0)
}
```

Fonction d'ordre supérieur et récursive (2, cont.)

```
func increment(_ x: Double) -> Double {
  return x + 1
}

func timesTwo(_ x: Double) -> Double {
  return x * 2
}

print(applyAll(2.0, [ increment, timesTwo, timesTwo, increment ]))
```

Fonction d'ordre supérieur et récursive (2, cont.)

```
func increment(_ x: Double) -> Double {
  return x + 1
}

func timesTwo(_ x: Double) -> Double {
  return x * 2
}

print(applyAll(2.0, [ increment, timesTwo, timesTwo, increment ]))
```

Affiche `13.0`.

Fonctions qui retournent une fonction

Fonctions qui retournent une fonction

On peut retourner comme résultat une fonction locale.

Toute les grandeurs définies dans le scope de la fonction perdurent.

On parle alors de closure.

Fonctions qui retournent une fonction

```
func labelPrinter(_ label: String) -> () -> () {
  func doIt() { print(label) }
  return doIt // label reste défini dans doIt!
}

let printBlah = labelPrinter("blah")

printBlah()

let printCoucou = labelPrinter("coucou")

printCoucou()
```

Affiche

```
blah
coucou
```

Fonctions qui retournent une fonction

```
func adder(_ delta: Double) -> (Double) -> Double {
  func f(x: Double) -> Double {
    return x + delta
  }

  return f // delta reste défini dans f!
}

let incrementBy45 = adder(45)

print(incrementBy45(40))
```

Fonctions qui retournent une fonction

```
func adder(_ delta: Double) -> (Double) -> Double {
  func f(x: Double) -> Double {
    return x + delta
  }

  return f // delta reste défini dans f!
}

let incrementBy45 = adder(45)

print(incrementBy45(40))
```

Affiche `85.0`.

Fonctions qui retournent une fonction

Il faut indiquer avec `@escaping` qu'une fonction doit rester définie. Ceci permet à Swift d'optimiser l'exécution quand cela n'est pas nécessaire.

```
func makeTwice(_ f: @escaping (Double) -> Double) -> (Double) -> Double {
  func h(x: Double) -> Double {
    return f(f(x))
  }
  return h
}

let phi = makeTwice(increment)

print(phi(2))
```

Il faut indiquer avec `@escaping` qu'une fonction doit rester définie. Ceci permet à Swift d'optimiser l'exécution quand cela n'est pas nécessaire.

```
func makeTwice(_ f: @escaping (Double) -> Double) -> (Double) -> Double {
  func h(x: Double) -> Double {
    return f(f(x))
  }
  return h
}

let phi = makeTwice(increment)

print(phi(2))
```

Affiche `4.0`.

Separation of concerns

Fonction à généraliser

Nous avons vu une fonction récursive pour tester que tous les éléments d'un tableau sont positifs.

```
func allPositive( x: [Int]  ) -> Bool {

  func allPositiveFrom( _ i: Int ) -> Bool {
    if i >= x.count  {
      return true
    } else {
      if x[i] < 0 {
        return false
      } else {
        return allPositiveFrom( i+1 )
      }
    }
  }

  return allPositiveFrom(0)
}
```

Pour tous: Généralisation

Nous pouvons généraliser cette fonction avec une fonction d'ordre supérieure qui prend en argument une propriété à tester:

```
func forAll( x: [Int], property: (Int) -> Bool ) -> Bool {

  func forAllFrom( _ i: Int ) -> Bool {
    if i >= x.count {
      return true
    } else {
      if !property( x[i] ) {
        return false
      } else {
        return forAllFrom( i+1 )
      }
    }
  }

  return forAllFrom(0)
}
```

Pour tous: Exemple d'utilisation (1)

Nous pouvons utiliser cette fonction d'ordre supérieur pour tester que tous les entiers d'un tableau sont positifs:

```
func isPositive( _ x: Int) -> Bool {
  return x >= 0
}

print(forAll( x: [ 2, 12, 3, 0 ], property: isPositive ))
```

Pour tous: Exemple d'utilisation (1)

Nous pouvons utiliser cette fonction d'ordre supérieur pour tester que tous les entiers d'un tableau sont positifs:

```
func isPositive( _ x: Int) -> Bool {
  return x >= 0
}

print(forAll( x: [ 2, 12, 3, 0 ], property: isPositive ))
```

Elle nous permet aussi d'écrire par exemple:

```
func allPositive( x: [Int]  ) -> Bool {
  return forAll(x, isPositive)
}
```

Pour tous: Exemple d'utilisation (2)

Et nous pouvons maintenant facilement écrire des fonctions similaires pour la parité:

```
func isEven( x: Int ) -> Bool {
  return x % 2 == 0
}

func allEven( x: [Int]  ) -> Bool {
  return forAll(x, isEven)
}
```

Prédicat

- En informatique on appelle prédicat une fonction qui retourne un Booléen.
- Le type d'un prédicat est donc `(T) -> Bool` où `T` est n'importe quel type du langage
- La notion de prédicat en logique du premier ordre est analogue.

À généraliser

```
func countPositive( _ x: [Int]  ) -> UInt {

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

  return countPositiveFrom(0,0)
}
```

Compter: généralisation

```
func count( _ x: [Int], _ p: (Int) -> Bool  ) -> UInt {

  func countFrom( _ i: Int, _ n: UInt  ) -> UInt {
    if i >= x.count  {
      return n
    } else {
      if p( x[i] ) {
        return countFrom( i+1, n+1 )
      } else {
        return countFrom( i+1, n )
      }
    }
  }

  return countFrom(0,0)
}
```

Compter: Exemples d'utilisation

```
// Compte les nombres positifs
func countPositive( _ x: [Int]  ) -> UInt {
  return count( x, isPositive )
}

// Compte les nombres pairs
func countEven( _ x: [Int]  ) -> UInt {
  return count( x, isEven )
}
```

Separation of concerns

"In computer science, **separation of concerns** is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern."

Wikipedia

Separation of concerns

	isPositive	isEven	isZero
forAll	allPositive	allEven	allZero
count	countPositive	countEven	countZero
exists	existsPositive	existsEven	existsZero

Fonctions anonymes

Fonction anonyme

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

Fonction anonyme

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

Par exemple, au lieu de:

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

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

Fonction anonyme

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

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 })
```

Fonction anonyme

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

Fonction anonyme

Arguments

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

Fonction anonyme

Arguments Résultat

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

Fonction anonyme

Arguments Résultat Corps

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

Fonction anonyme

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
  }
}))
```

Lorsque le type de retour et/ou des arguments peuvent être inférés, leur déclaration est facultative. Et s'il n'y a qu'une expression, le `return` est facultatif.

```
 1> let f = { (i: Int) -> Bool in return i != 0 }
f: (Int) -> Bool =
 2> let f = { i -> Bool in return i != 0 }
f: (Int) -> Bool =
 3> let f = { i in return i != 0 }
f: (Int) -> Bool =
 4> let f = { i in i != 0 }
f: (Int) -> Bool =
```

Logique des prédicats

Le code suivant permet de déterminer si tous les nombres d'un tableau sont positifs:

```
func allPositive( x: [Int]  ) -> Bool {
  return forAll(x, isPositive)
}
```

Comment répondre à la question: existe-t'il au moins un nombre positif ?

```
func exitsPos( x: [Int]  ) -> Bool
```

La relation suivante nous permet de transformer un quantificateur universel (`forAll`) en quantificateur existentiel (`exists`):

$$(\exists x \in \mathscr{X}, P(x)) \Leftrightarrow \neg\,(\forall x \in \mathscr{X}, \neg P(x))$$

La relation suivante nous permet de transformer un quantificateur universel (`forAll`) en quantificateur existentiel (`exists`):

$$(\exists x \in \mathcal{X}, P(x)) \Leftrightarrow \neg\,(\forall x \in \mathcal{X}, \neg P(x))$$

```
func isPositive( _ i: Int) -> Bool {
  return i >= 0
}

func exists( x: [Int], property: (Int) -> Bool ) -> Bool {
  return !forAll( x: x, property: { i in !property(i) } )
}

print(forAll( x: [ 1, 4, 0, 4 ], property: isPositive ))
print(forAll( x: [ -1, 4, -2, -4 ], property: isPositive ))
print(exists( x: [ -1, 4, -2, -4 ], property: isPositive ))
```

Exemples d'utilisations de fonctions anonymes

Point le plus proche

```
struct Point {
  let x, y: Double
}

func closest(p: Point, cloud: [Point], distance: (Point, Point) -> Double) -> Point
```

Point le plus proche (cont.)

```
func closest(p: Point, cloud: [Point], distance: (Point, Point) -> Double) -> Point {
  func closest(from: Int, best: Int, bestDistance: Double) -> Point {
    if from < cloud.count {
      let d = distance(p, cloud[from])
      if d < bestDistance {
        return closest(from: from + 1, best: from, bestDistance: d)
      } else {
        return closest(from: from + 1, best: best, bestDistance: bestDistance)
      }
    } else {
      return cloud[best]
    }
  }

  return closest(from: 1, best: 0, bestDistance: distance(p, cloud[0]))
}
```

Point le plus proche (cont.)

```
func euclideanDistance(p: Point, q: Point) -> Double {
  func sq( _ x: Double) -> Double { return x*x }
  return sqrt(sq(p.x - q.x) + sq(p.y - q.y))
}

q = closest(
  p: Point(x: 1.0, y: 2.0),
  cloud: [ Point(x: 1.1, y: 1.9), Point(x: 5.3, y: 11.2) ],
  distance: euclideanDistance
)
```

Point le plus proche (cont.)

```
let q = closest(
  p: Point(x: 1.0, y: 2.0),
  cloud: [ Point(x: 1.1, y: 1.9), Point(x: 5.3, y: 11.2) ],
  distance: { (p: Point, q: Point) -> Double in
    func sq( _ x: Double) -> Double { return x*x }
    return sqrt(sq(p.x - q.x) + sq(p.y - q.y))
  }
)
```

Gestion des prix

```
struct Item {
  let name: String
  let unitPrice: UInt
  let amount: UInt
}

func updatePrice( item: Item, priceCalculator: (Item) -> UInt ) -> Item {
  let newPrice = priceCalculator(item)
  return Item(
    name: item.name,
    unitPrice: newPrice,
    amount: item.amount
  )
}
```

Gestion des prix (cont.)

```
let sausages = Item(
  name: "Sausage",
  unitPrice: 40,
  amount: 25
)

let sausagesWithRebate = updatePrice(
  item: sausages,
  priceCalculator: { item in
    if item.amount >= 20 {
      return item.unitPrice * 95 / 100
    } else {
      return item.unitPrice
    }
  }
)

print(sausages)
print(sausagesWithRebate)
```

Affiche

```
Item(name: "Sausage", unitPrice: 40, amount: 25)
Item(name: "Sausage", unitPrice: 38, amount: 25)
```

Gestion des prix (cont.)

```
func priceUpdater(priceCalculator: @escaping (Item) -> UInt ) -> (Item) -> Item {
  return { item in
    let newPrice = priceCalculator(item)
    return Item(
      name: item.name,
      unitPrice: newPrice,
      amount: item.amount
    )
  }
}
```

Gestion des prix (cont.)

```
let rebater = priceUpdater(
  priceCalculator: { item in
    if item.amount >= 20 {
      return item.unitPrice * 95 / 100
    } else {
      return item.unitPrice
    }
  }
)

print(rebater(sausages))
```

Affiche

```
Item(name: "Sausage", unitPrice: 38, amount: 25)
```

FIN