Design patterns fonctionnels

Le “fonctionnelle” de programmation fonctionnelle

Jean-Michel Begon

Université de Liège

23 mars 2017

`http://www.montefiore.ulg.ac.be/~jmbegon/`

Plan

1. Liste d'arguments
   - Taille variable
   - Au moins un argument
   - Valeurs par défaut
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
   - Closure
   - Mind blowing time
   - Curryfication
4. Pattern
   - Constructeur/factory
   - Décorateur
   - Store-passing style (SPS)
     - Itérateur
   - Continuation et CPS
   - Autres design patterns
5. Conclusion

Plan

1. Liste d'arguments
   - Taille variable
   - Au moins un argument
   - Valeurs par défaut
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
4. Pattern
5. Conclusion

Liste d'arguments : taille variable

Objectif : Permettre l'utilisation d'un nombre variable d'arguments.

Exemples : `list`, `+`, `and`, `...`

```
(define list*
  (lambda args
    (if (null? args) '()
        (cons (car args) (apply list* (cdr args))))))
```

Quel est l'intérêt de `apply` ?

`(apply f a b ... z (list A B .. Z))`
équivaut à
`(f a b ... z A B ... Z)`

Liste d'arguments : au moins un argument

Objectif : Imposer d'avoir au moins un argument.

```
(define min+
  (lambda (ca . cd)
    (if (null? cd) ca
        (let ((m (apply min+ cd)))
          (if (< ca m) ca m)))))

(define list+
  (lambda (ca . cd)
    (if (null? cd) (cons ca '())
        (cons ca (apply list+ cd)))))
```

Liste d'arguments : valeurs par défaut

Objectif : Eviter de préciser la valeur d'un argument qui est souvent la même.

Exemple : La base *e* est la plus utilisée pour un logarithme.

```
(define log&
  (lambda (x [base (exp 1)])
    (/ (log x) (log base))))

(log& 16)
--> 2.772588722239781

(log& 16 2)
--> 4.0
```

Liste d'arguments : arguments par mot-clé

```
(define log#
  (lambda (x #:base base-value)
    (/ (log x) (log base-value))))

(define log#&
  (lambda (x #:base [base-value (exp 1)])
    (/ (log x) (log base-value))))
```

```
(log# 16 #:base 2)
--> 4.0

(log# 16)
--> error
```

```
(log#& 16 #:base 2)
--> 4.0

(log#& 16)
--> 2.772588722239781
```

Plan

1. Liste d'arguments
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
4. Pattern
5. Conclusion

Fonction en entrée

Objectif : Découpler les données des traitements à effectuer.

Exemples : `MapReduce`

- Calculer la norme d'un vecteur représenté par une liste de nombres.
- Calculer la plus petite valeur des cosinus d'une liste de nombres.

Problème : Les deux calculs suivent le même schéma.

Autre exemple : Trier des listes d'objets complexes.

Fonction en entrée — Mauvaise solution

Fournir deux implémentations

```
(define norm
  (lambda (ls)
    (if (null? ls) 0
        (+ (* (car ls) (car ls)) (norm (cdr ls))))))

(define mincos
  (lambda (ls)
    (if (null? ls) +inf.f
        (let ((min-cdr (mincos (cdr ls)))
              (cos-car (cos (car ls))))
          (if (< cos-car min-cdr) cos-car
              min-cdr)))))
```

Que se passe-t-il lorsqu'on doit implémenter une nouvelle fonction ?

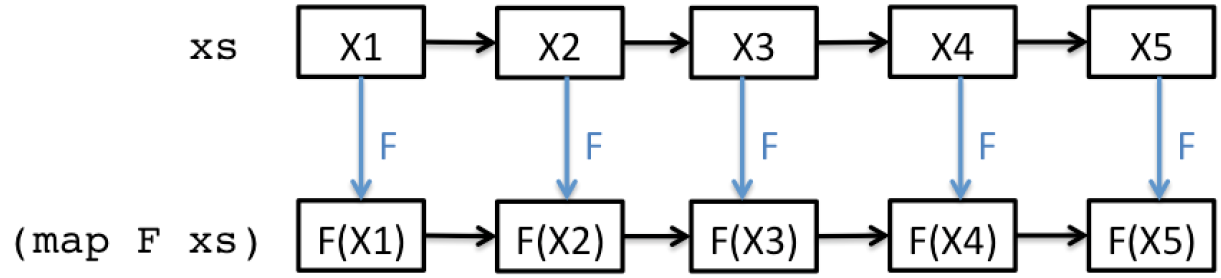
Fonction en entrée — Meilleure solution

Découpler les responsabilités en utilisant des fonctions en entrée

```
(define norm*
    (lambda (ls)
      (reduce + 0 (map (lambda (x) (* x x)) ls))))

(define mincos*
  (lambda (ls)
    (reduce (lambda (x y) (if (< x y) x y))
            +inf.f
            (map cos ls))))
```

Map



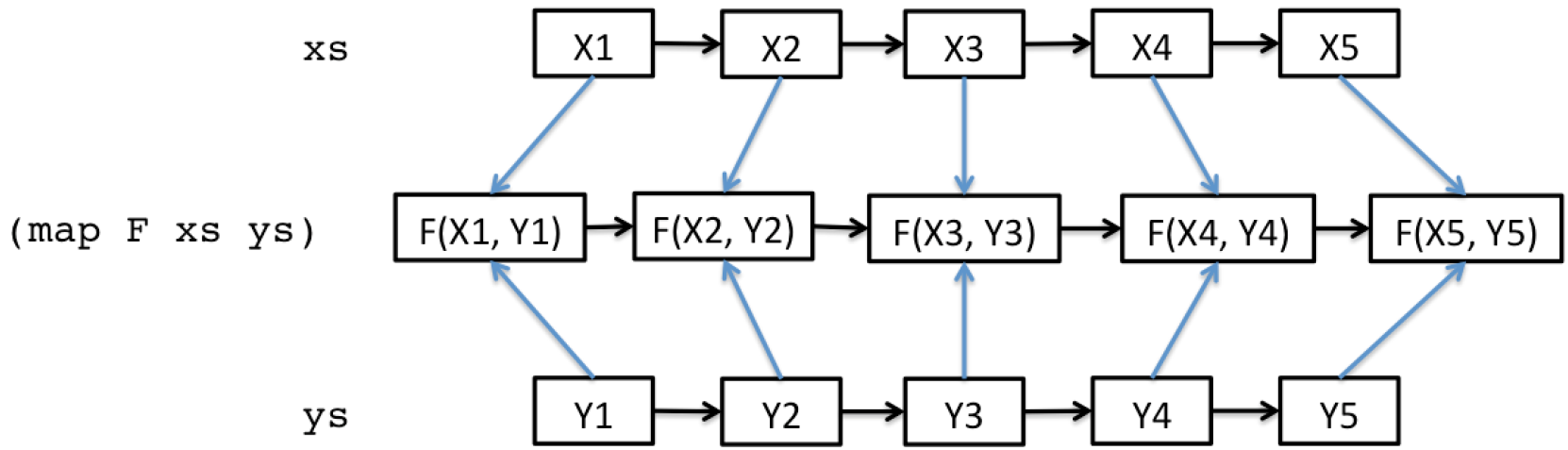


Figure: `map`

Reduce

```
(define reduce
  (lambda (f b ls)
    (if (null? ls) b
        (reduce f
                (f b (car ls))
                (cdr ls)))))
```

Aussi connu sous le nom de `fold`.
Les fonctions `foldl` et `foldr` font cependant débat dans le monde de la programmation fonctionnelle :

(`http://stackoverflow.com/questions/8778492/why-is-foldl-defined-in-a-strange-way-in-racket`)

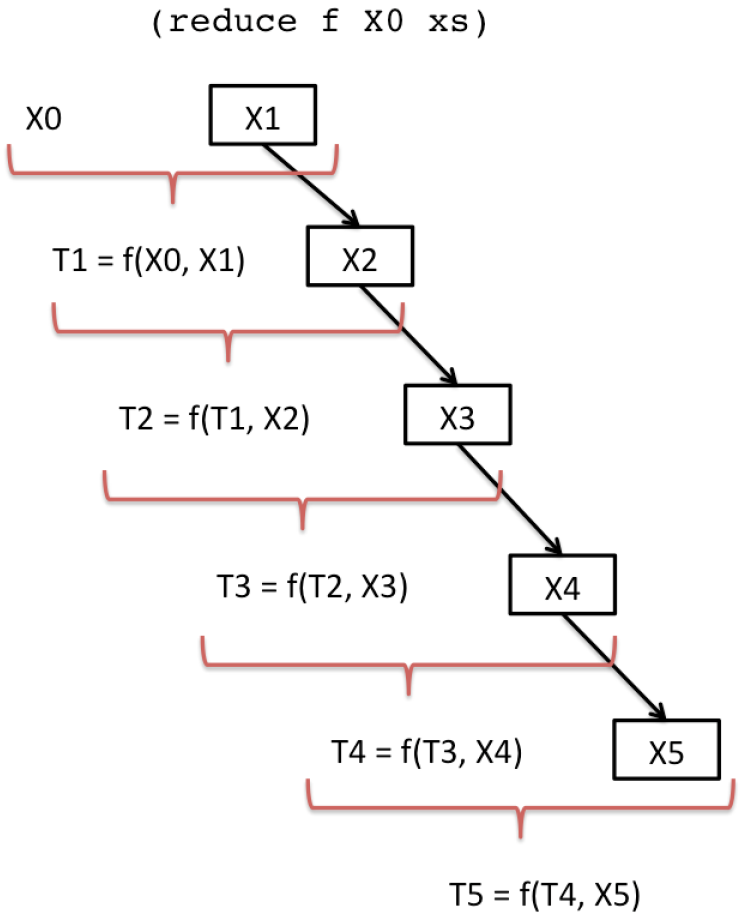


Figure: `reduce`

Plan

1. Liste d'arguments
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
   - Closure
   - Mind blowing time
   - Curryfication
4. Pattern
5. Conclusion

Closure

```
Make-incrementer(X)
1   Increment(Y)
2       return X + Y
3   return Increment
```

```
1   Plus2 = Make-incrementer(2)
2   Plus2(5) // return 7
```

Lorsque la fonction Increment est retournée, elle garde la trace de valeur de X à utiliser. C'est la closure.

On dit que la valeur X est dans la fermeture de Increment.

Les paires pointées ne sont pas nécessaires !

```
(define cons
    (lambda (x y)
        (lambda (f) (f x y))))

(define car
    (lambda (g)
        (g (lambda (p q) p))))

(cons 'a 'b)
((lambda (x y) (lambda (f) (f x y))) 'a 'b)
(lambda (f) (f 'a 'b))

(car (cons 'a 'b))
((lambda (g) (g (lambda (p q) p))) (lambda (f) (f 'a 'b)))
((lambda (f) (f 'a 'b)) (lambda (p q) p)
((lambda (p q) p) 'a 'b)
```

Curryfication

La curryfication (anglais : *currying*) est le fait de passer d'une fonction de n ($n > 1$) arguments (1) à une fonction d'un argument qui renvoie une nouvelle fonction à un argument, ... (2)

$$+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R} \quad (1)$$

$$+ + : \mathbb{R} \to \mathbb{R} \to \mathbb{R} \quad (2)$$

Ce mécanisme est implicite en Haskell. On parle d'application partielle.

Plan

1. Liste d'arguments
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
4. Pattern
   - Constructeur/factory
   - Décorateur
   - Store-passing style (SPS)
     - Itérateur
   - Continuation et CPS
   - Autres design patterns
5. Conclusion

Constructeur/factory

Objectif : paramétriser une fonction.

Exemple : appliquer des fonctions linéaires différentes

Problème : redondance du code.

Constructeur/factory — Mauvaise solution

Définir une fonction par base :

```
(define 2x+0
  (lambda (x)
    (+ (* 2 x) 0)))

(define 7x+10
  (lambda (x)
    (+ (* 7 x) 10)))

(define x+3
  (lambda (x)
    (+ (* 1 x) 3)))
```

Constructeur/factory — Meilleure solution

Définir une fonction qui prend les paramètres en entrée et qui renvoie la fonction paramétrée :

```
(define ax+b
  (lambda (a b)
    (lambda (x)
      (+ (* a x) b))))

(define 2x+0 (ax+b 2 0))
(define 7x+10 (ax+b 7 10))
(define x+3 (ax+b 1 3))
```

Décorateur

Objectif :	augmenter une fonction sans changer sa sémantique.
Exemple :	système de logs.
Problème :	découpler la fonction du mécanisme de logging.
Remarques :	à la limite de la philosophie déclarative pour les utilisations les plus intéressantes (timer de fonction, logging, etc.)

Décorateur

```
(define arg-deco
    (lambda (f)
        (lambda args
            (begin (display args) (newline)
                   (apply f args)))))

(define fib-a
  (lambda (n a b)
    (if (zero? n) a
        (fib-a (- n 1) b (+ a b)))))

((arg-deco fib-a) 10 0 1)
--> 55
::: (10 0 1)
```

arg-deco renvoie une fonction qui donne le même résultat que la fonction en entrée mais qui imprime les arguments de l'appel **initial**.

Store

Objectif : gérer un objet qui représente l'état d'un système.

Exemples : représenter la position d'un personnage mobile dans un jeu, simuler une machine à états, ...

Problème : les entités ont un état mais les objets en mémoire sont immuables.

Store — Solution

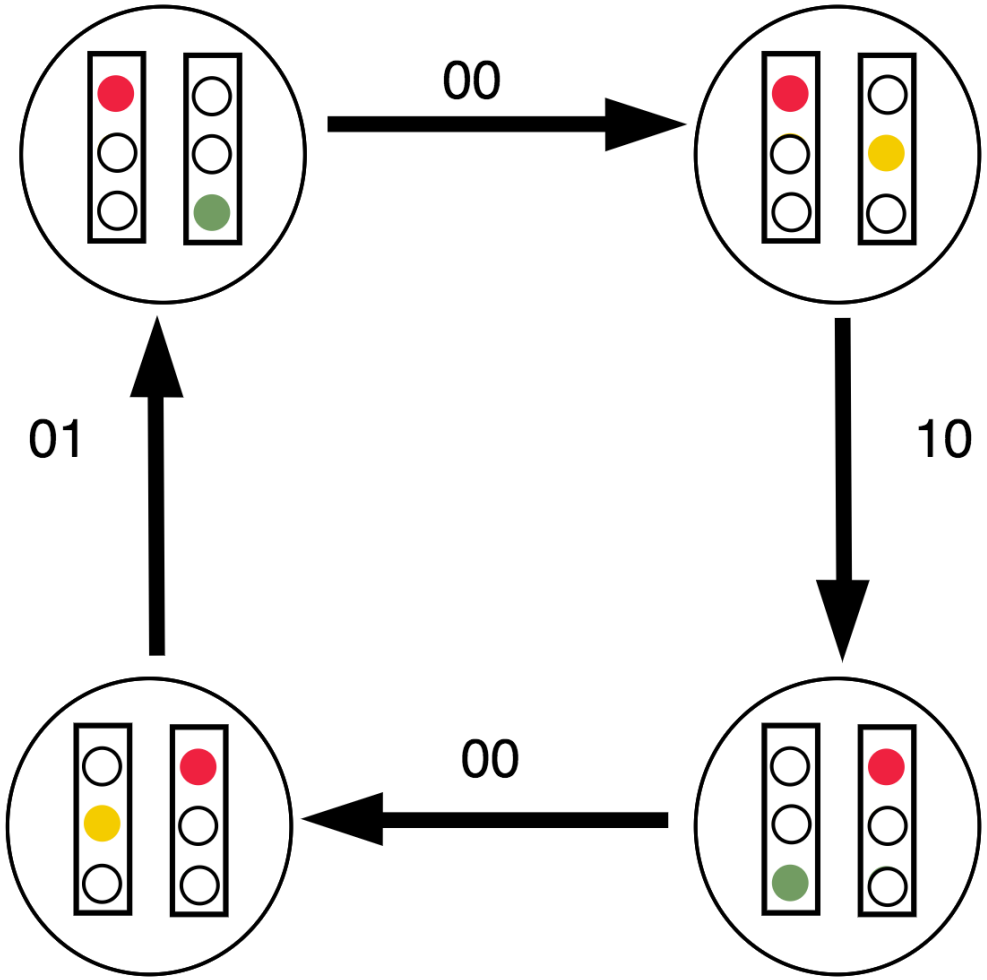
- L'état est représenté par une structure : un `store`.
- Un nouveau `store` est reconstruit à chaque modification.

```
; Structure
(define cons-position cons)
(define position.x car)
(define position.y cdr)

(define up
  (lambda (position)
    (cons-position (+ 1 (position.x position))
                   (position.y position))))

(up (cons-position 0 0))
--> '(1 . 0)
```

Store-passing style



Problème : il y a une valeur de retour associée à chaque transition.

Store-passing style — solution

La fonction prend le `store` en entrée et renvoie le nouveau `store` avec les autres valeurs de retour.

```
(define change-lights
  (lambda (light-pair)
    (cond ((and (red? (car light-pair))
                (green? (cdr light-pair)))
           (cons (cons 'red 'yellow) (cons 0 0)))
          ((and (red? (car light-pair))
                (yellow? (cdr light-pair)))
           (cons (cons 'green 'red) (cons 1 0)))
          ((and (green? (car light-pair))
                (red? (cdr light-pair)))
           (cons (cons 'yellow 'red) (cons 0 0)))
          (else (cons (cons 'red 'green) (cons 0 1))))))

(change-lights (cons 'red 'green))
--> '((red . yellow) 0 . 0)
```

Store-passing style — aperçu

`Store`	Aucune autre valeur de retour	Autres valeurs de retour
Explicite	`Store`	`Store`-passing style (SPS)
Implicite	Simple Closed SPS	Closed SPS

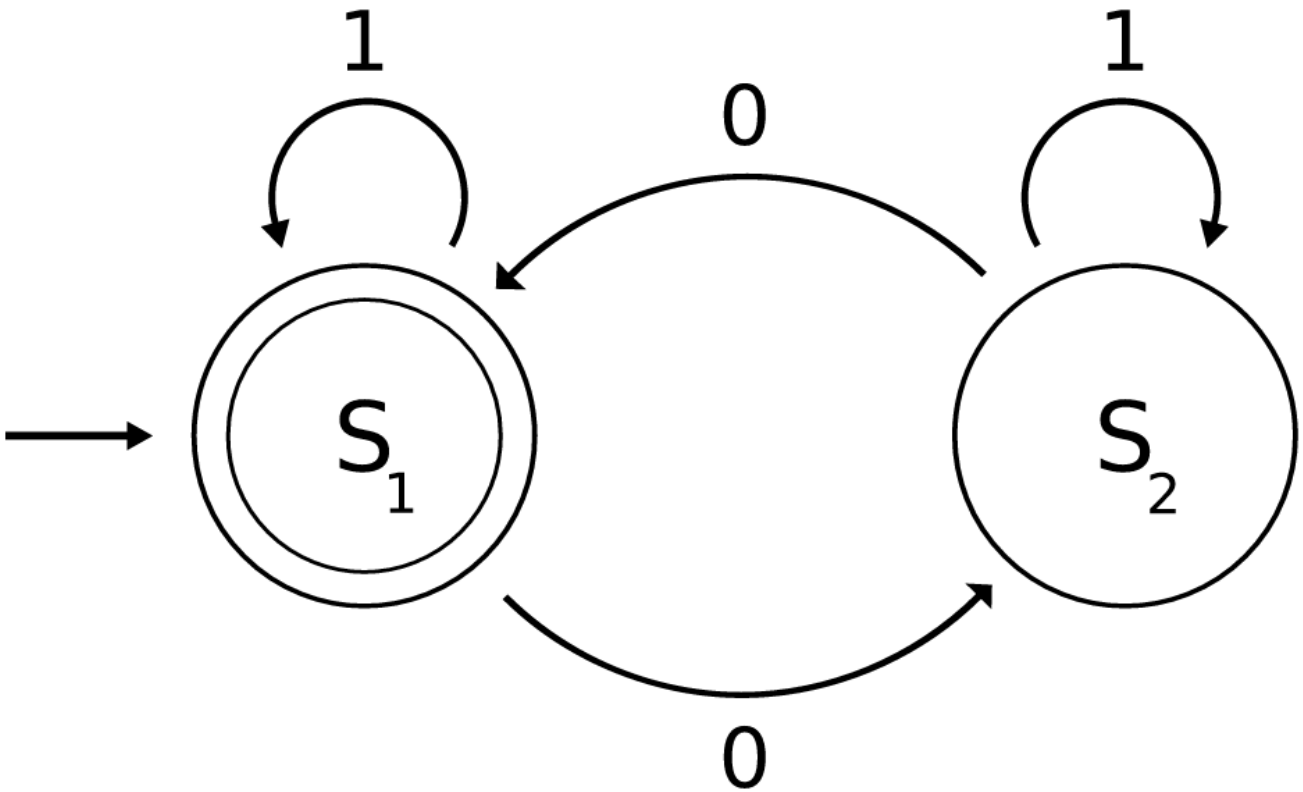
Un `store` est implicite si

- c'est une fonction, ou
- s'il est dans la fermeture d'une fonction plutôt que d'être un de ses arguments.

Si la fonction du Closed SPS ne prend pas d'argument, on parle d'itérateur.

(Simple) closed SPS

Modéliser un automate fini déterministe :



Il y a-t-il un nombre pair de zéros ?

(Simple) closed SPS — Solution

```
(define fsm-S1
  (lambda (b)
    (cond
      ((null? b) #t)
      ((zero? b) fsm-S2)
      (else fsm-S1))))
```

```
(define fsm-S2
  (lambda (b)
    (cond
      ((null? b) #f)
      ((zero? b) fsm-S1)
      (else fsm-S2))))
```

```
(define fsm
  (lambda (state ls)
    (if (null? ls) (state '())
        (fsm (state (car ls)) (cdr ls)))))
```

```
(fsm fsm-S1 (list 0 0 1 0))
--> #f
(fsm fsm-S1 (list 0 0 1 0 0))
--> #t
```

Closed SPS — Itérateur

Objectif : générer une suite de valeurs sans devoir calculer toute la séquence

Exemples : itérer sur les éléments d'une liste, d'un arbre, un automate fini, etc. Générer les suites de la factorielle, de Fibonacci, etc.

Problème : les suites peuvent être infinies ou les calculs trop longs, on veut éviter de recalculer des résultats.

Remarque : s'inscrit dans le pattern *lazy*.

Comment ? *closed SPS*.

Closed SPS — Itérateur

Exemple de la factorielle :

```
(define fact-it
  (lambda (n acc)
    (lambda ()
      (cons acc (fact-it (+ n 1)
                         (* (+ n 1) acc))))))

(define it.n
  (lambda (it n)
    (if (zero? n) (car (it))
        (it.n (cdr (it)) (- n 1)))))
```

```
(it.n (fact-it 0 1) 0)
--> 1
(it.n (fact-it 0 1) 1)
--> 1
(it.n (fact-it 0 1) 2)
--> 2
```

```
(it.n (fact-it 0 1) 3)
--> 3
(it.n (fact-it 0 1) 6)
--> 720
(it.n (fact-it 0 1) 10)
--> 3628800
```

Closed SPS — Itérateur

Exemple des feux de signalisation :

```
(define red-green
  (lambda ()
    (cons red-yellow (cons 0 0))))

(define red-yellow
  (lambda ()
    (cons green-red (cons 1 0))))

(define green-red
  (lambda ()
    (cons yellow-red (cons 0 0))))

(define yellow-red
  (lambda ()
    (cons red-green (cons 0 1))))
```

Continuation et CPS

Continuation : représentation (abstraite) du *control flow*, généralement sous la forme d'une fonction.

Control flow : l'ordre dans lequel les instructions/fonctions doivent être exécutées/évaluées.

CPS : fait de construire/passer explicitement la continuation au travers de plusieurs appels de fonctions.

CPS vs SPS :

- Le `store` représente la suite de calculs à effectuer.

Du point de vue impératif, les appels récursifs servent à construire la fonction qui sera appliquée au cas de base.

- La fonction CPS est donc *tail-recursive*. Néanmoins, la construction récursive de la continuation implique de sauvegarder toutes les fermetures des fonctions intermédiaires.

Puisque la fonction n'est effectivement appliquée que lorsqu'elle atteint son cas de base, le CPS est une forme de *lazy programming*.

CPS — Exemple

```
; (k (apply + ls))
(define +&
  (lambda (ls k)
    (if (null? ls) (k 0)
        (+& (cdr ls)
            (lambda (x) (k (+ x (car ls))))))))

(+& '(1 2 3 4) (lambda (x) x))
--> 10
```

Autres design patterns

Il existe d'autres constructions spécifiques à la programmation fonctionnelle :

- Functors
- Applicative
- Monads
- ...

Mais celles-ci ne seront pas abordées dans le cadre de ce cours.

Plan

1. Liste d'arguments
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
4. Pattern
5. Conclusion

Conclusion

1. Liste d'arguments
   - Taille variable
   - Au moins un argument
   - Valeurs par défaut
2. Fonction d'ordre supérieur — Inputs
3. Fonction d'ordre supérieur — Outputs et Closures
   - Closure
   - Mind blowing time
   - Curryfication
4. Pattern
   - Constructeur/factory
   - Décorateur
   - Store-passing style (SPS)
     - Itérateur
   - Continuation et CPS
   - Autres design patterns
5. Conclusion