Programmation fonctionnelle

semestre 3

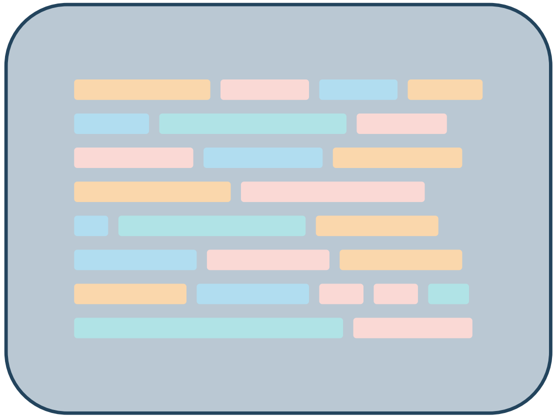
Thomas Baruchel
Collège SPEE
thomas.baruchel@univ-pau.fr

aucun droit réservé

Introduction

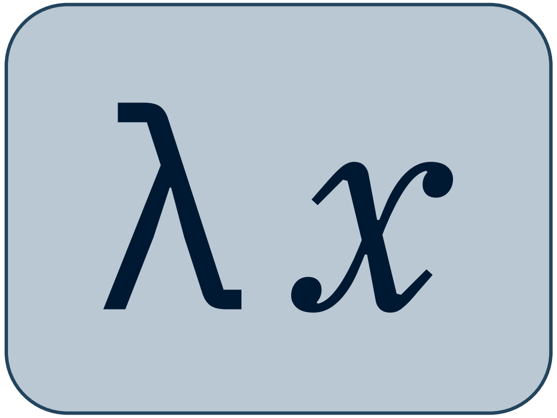
Grands paradigmes de la programmation # 1

Quelques grands paradigmes de programmation :

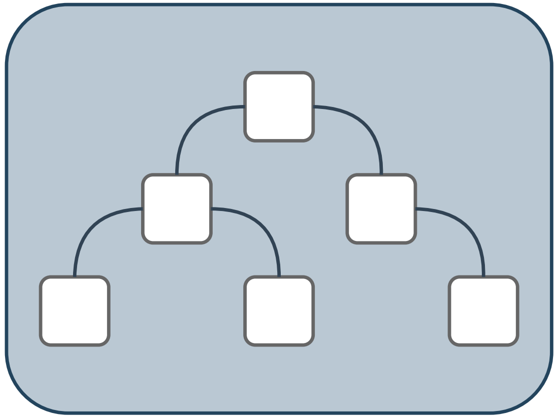


programmation impérative

(ex : C)



programmation fonctionnelle



programmation orientée objet

De nombreux autres paradigmes existent (logique, parallèle, événementiel, etc.)

Présentation (1)

Un ensemble de principes :

→ les fonctions comme objets de première classe
→ importance de la récursivité
→ recours aux listes (en lien avec la récursivité)
→ privilégier les expressions aux instructions
→ mettre l'accent sur le but du calcul plutôt que sur la méthode

Ces principes sont mis en œuvre :

→ dans certains langages privilégiés (Scheme, Haskell, etc.)
→ dans un certain style de programmation

+ OCaml

Un modèle de calcul sous-jacent : ***le lambda-calcul***

λ

Présentation (2)

Intérêts de la programmation fonctionnelle :

→ comportement prédictible des fonctions
→ élimination de certaines sources de bugs
→ forte compatibilité avec la modularité
→ forte compatibilité avec le parallélisme
→ mise en œuvre aisée de tests
→ des techniques spécifiques de preuve de correction

Inconvénients de la programmation fonctionnelle :

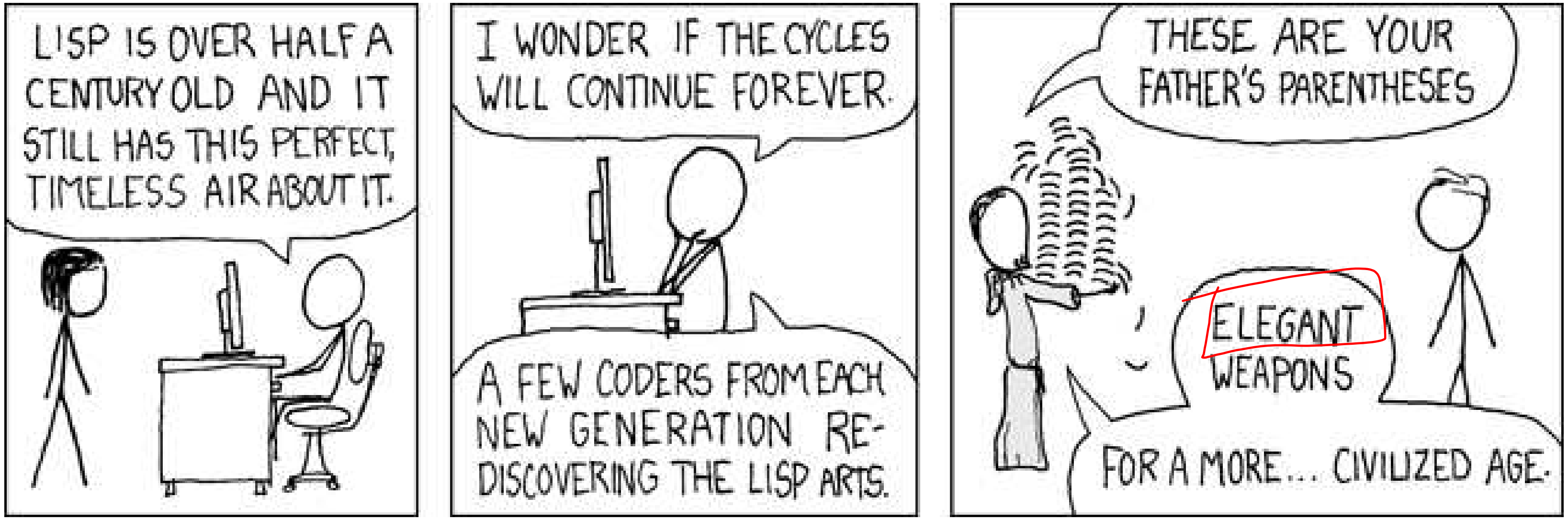
→ utilisation parfois non optimale de la mémoire
→ nombre limité de modules et bibliothèques
→ taille modeste des communautés de programmeurs

Le cours s'appuie sur le langage *Scheme*

→ un dialecte du langage Lisp

→ créé au cours des années 1970

→ de très nombreuses implémentations



Source : https://xkcd.com/297

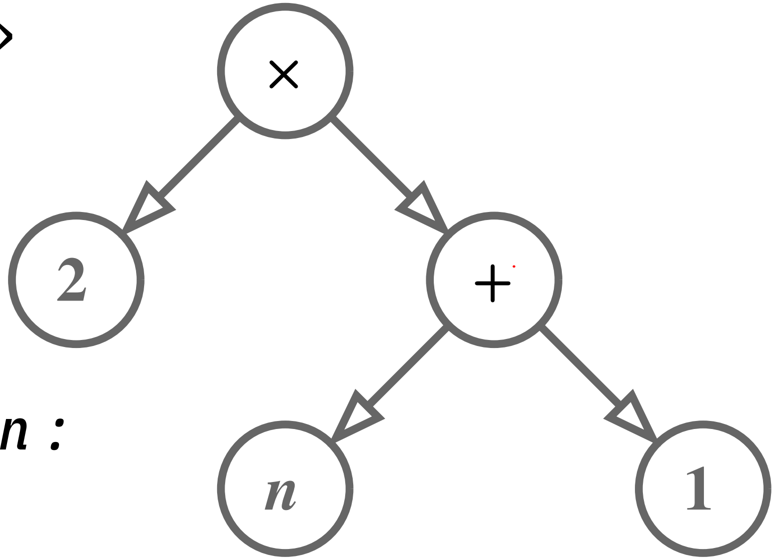
Découverte du langage Scheme

On pourra utiliser l'IDE *DrRacket* dans le cours et les TP

→ analyse du code pour une addition simple
→ analyse d'une opération arithmétique complexe
→ analyse d'une comparaison simple
→ analyse d'une expression booléenne complexe

Repérer la structure des S-expressions : → Lisp

→ notation polonaise « de Cambridge »
→ recours aux parenthèses
→ homogénéité totale de la syntaxe
(homoiconicité du langage)



Ci-contre, une représentation de la S-expression :

```
(* (2 (+ n 1)))
```

$\boxed{A}$ $((2 < 3) \text{ or } (4 < 3))$

et $(6 > 5)$ $\boxed{B}$

(and (or (< 2 3) (< 4 3))
(> 6 5))

$$\underbrace{2+3}-\underbrace{4}$$

Trace

(− (+ 2 3) 4)

(− 5 4)

1

Structures simples en Scheme # 6

Spécificité d'un test en Scheme :

```
(if (< 1 2) 3 4)
```

→ un test constitue une expression
→ l'expression complète possède une *valeur*
→ le test peut appartenir à une expression plus large

Analyser le code de quelques fonctions simples

→ analyse du code de la fonction *double*
→ analyse du code de la fonction *carré*
→ analyse du code de la fonction *factorielle*
→ analyse de la version naïve de la fonction *fibonacci*

$$(\text{if } \underbrace{\langle \text{condition} \rangle}_{\text{booléen}} \; \underbrace{\langle \text{then} \rangle}_{\text{expression 1}} \; \underbrace{\langle \text{else} \rangle}_{\text{expression 2}})$$

Factorielle :

Définition

$n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n$

$\boxed{0! = 1}$

ex : $3! = 6 = 1 \times 2 \times 3$

$6! = 720$

$= 1 \times 2 \times 3 \times 4 \times 5 \times 6$

$$n! = \begin{cases} 1 & (\text{si } n = 0) \\ \underline{n} \times (n-1)! & (\text{sinon}) \end{cases}$$

condition d'arrêt

Analyse

↳ Deux cas distincts

Ex : $5!$

$= 5 \times (5-1)!$

$= 5 \times (4 \times 3!)$

$= 5 \times (4 \times (3 \times 2!))$

Calcul de Fibonacci(n)

Suite de F. :

$0, 1, 1, 2, 3, 5, 8, \dots$

$$\begin{cases} F_0 = 0, \quad F_1 = 1 \\ F_{n+2} = F_n + F_{n+1} \end{cases}$$

ou :

$$\begin{cases} F_0 = 0, \quad F_1 = 1 \\ F_n = F_{n-2} + F_{n-1} \quad (\text{pour } n \geq 2) \end{cases}$$

ou

$$\begin{cases} F_n = n \quad (\text{si } n < 2) \\ F_n = F_{n-2} + F_{n-1} \quad (\text{si } n \geq 2) \end{cases}$$

Voici les fonctions qui viennent d'être examinées :

```scheme
(define (double n)
  (* 2 n))

(define (carre x)
  (* x x))            ; ou (expt x 2)

(define (factorielle n)
  (if (< n 2)
    1
    (* n (factorielle (- n 1)))))

(define (fibonacci n)
  (if (< n 2)
    n
    (+ (fibonacci (- n 2))
       (fibonacci (- n 1)))))
```

Définitions

Les définitions s'effectuent de la façon suivante :

```
; définition d'une variable
(define variable valeur)

; définition d'une fonction
(define (fonction arg1 arg2 etc.)
  ( ; corps de la fonction
  ))
```

Les variables et les fonctions restent donc homogènes.

On évite autant que possible de modifier la valeur d'une variable !

→ particularité de la programmation fonctionnelle
→ on définit ainsi plutôt des constantes
→ il reste cependant possible de le faire avec `set!`
(noter la sémantique du point d'exclamation)

Exemple du PGCD

La programmation fonctionnelle évite, si possible, les boucles :

→ les boucles impliquent des modifications de valeurs

```
; Calcul récursif du PGCD de deux entiers
;    - aucune boucle n'est utilisée ici
;    - on ne modifie la valeur d'aucune variable
(define (pgcd a b)
  (if (zero? b) a
    (pgcd b (modulo a b))))
```

plus petit que b ⇒ condition d'arrêt

→ l'appel récursif remplace ici la traditionnelle boucle `while`

→ la définition donne ici une sorte de règle de réécriture

→ on privilégie l'idée (mathématique) au mécanisme de calcul (le code ressemble ici à une définition du PGCD)

$1\,1\,0\,1$

$/2 \rightarrow 1\,1\,0$

$$13_{(10)} = \underbrace{110\cancel{1}}_{\text{4 chiffres binaires}}{}_{(2)}$$

$$bl(13) = 4$$

Analyse

$bl(13) = 1 + bl(13//2)$

$$bl(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ 1 + bl(n//2) & \end{cases}$$

Exemple du nombre de bits

Voici un second exemple :

```
; calcul du nombre de bits nécessaires
; pour l'écriture binaire du nombre n
(define (bitlength n)
  (if (zero? n) 0
    (+ 1 (bitlength (quotient n 2)))))
```

L'évaluation est ici assimilable à un système de réécriture :

```
  (bitlength 19)
↪ (+ 1 (bitlength 9))
↪ (+ 1 (+ 1 (bitlength 4)))
↪ (+ 1 (+ 1 (+ 1 (bitlength 2))))
↪ (+ 1 (+ 1 (+ 1 (+ 1 (bitlength 1)))))
↪ (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (bitlength 0))))))
↪ (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0)))))
```

Évaluation séquentielle

L'évaluation séquentielle reste possible avec **`begin`** :

```
(define (boum n)
  (if (zero? n)
    (begin
      (display "0 Boum !")
      (newline))
    (begin
      (display n)
      (display " ...")
      (newline)
      (boum (- n 1)))))
```

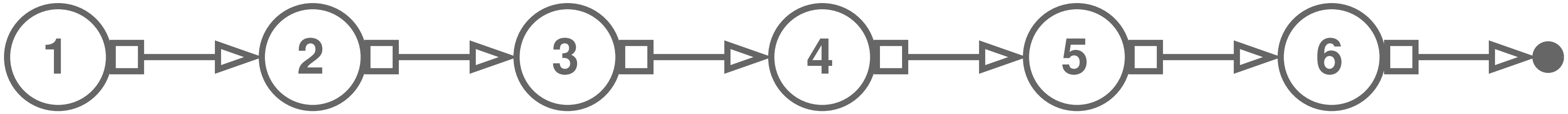
La sous-expression prend la valeur de la dernière évaluation.
(Dans l'exemple ci-dessus, la valeur est sans pertinence.)

Les langages fonctionnels privilégient les listes chaînées :

```
(define maliste '(1 2 3 4 5 6))
```

(L'apostrophe empêche ici la parenthèse d'être évaluée !)

La liste définie ci-dessus peut être représentée comme ceci :



Voici une autre représentation, typique des dialectes Lisp :

```
(1 . (2 . (3 . (4 . (5 . (6 . ()))))))
```

où l'on voit qu'*en dehors de la liste vide,*

→ toute liste est assimilable à un couple

→ le premier élément d'un couple donne un élément de la liste

→ le second élément d'un couple pointe vers le reste de la liste

Listes (2)

On manipule les listes (chaînées) grâce à trois fonctions :

→ extraction du premier élément d'une liste

```
(car '(1 2 3 4 5 6))
; ↳ est évalué à 1
```

→ extraction du « reste » de la liste

```
(cdr '(1 2 3 4 5 6))
; ↳ est évalué à (2 3 4 5 6)
```

→ ajout d'un élément en tête de la liste

```
(cons 0 '(1 2 3 4 5 6))
; ↳ est évalué à (0 1 2 3 4 5 6)
```

Ces trois fonctions ont un coût indépendant de la taille de la liste.

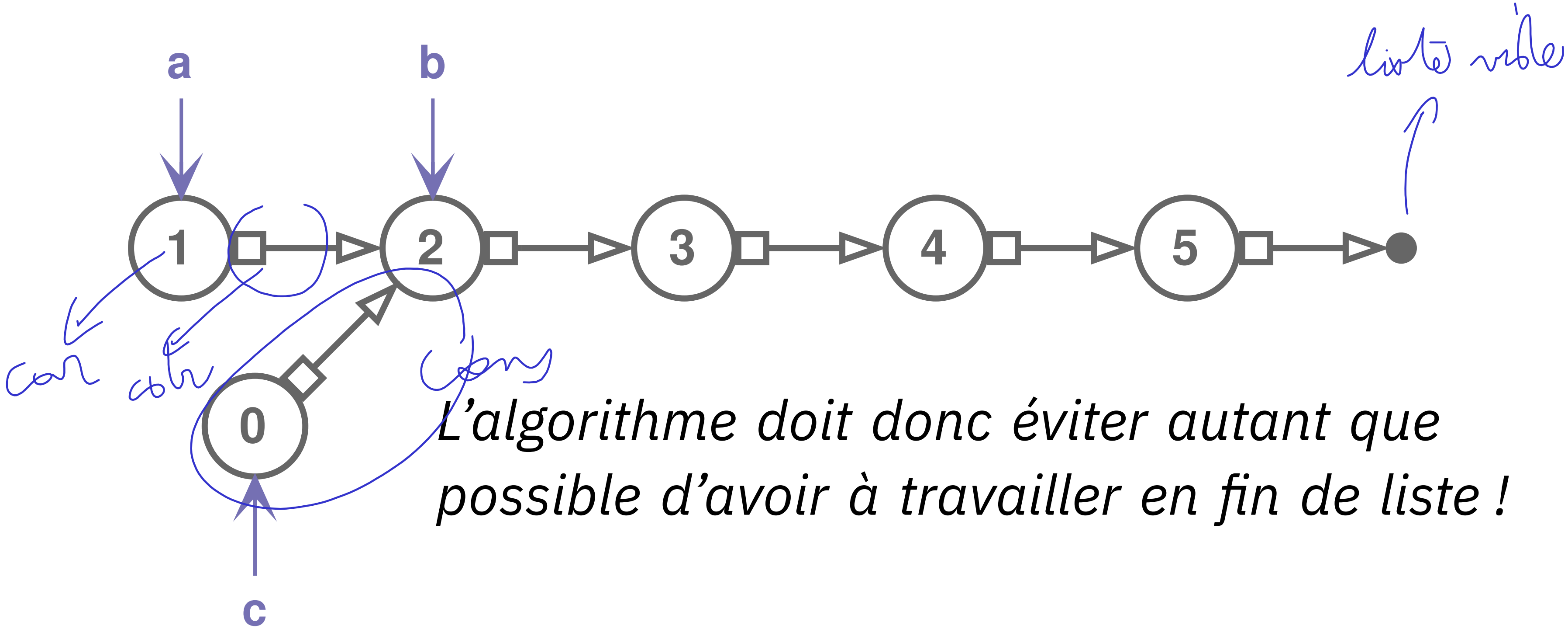
Aucune de ces trois fonctions ne modifie « en place » la liste.

Aucune de ces trois fonctions n'effectue de copie de liste.

Le travail en tête de liste ne donne lieu à aucune copie de liste.

```
(define a '(1 2 3 4 5))
(define b (cdr a))
(define c (cons 0 b))
```

aboutit à la construction suivante en mémoire :



L'algorithme doit donc éviter autant que possible d'avoir à travailler en fin de liste !

On construit fréquemment des listes « à l'envers » :

```scheme
; liste des entiers de 0 à n-1 (liste à l'envers)
(define (range n)
  (if (zero? n) '()
    (cons (- n 1) (range (- n 1)))))
```

Ici, `(range 10)` donne `(9 8 7 6 5 4 3 2 1 0)`.

On peut cependant retourner la liste à la fin de la construction :

```scheme
; liste des entiers de 0 à n-1 (liste à l'endroit)
(define (range* n) (reverse (range n)))
```

où `(range* 10)` donne `(0 1 2 3 4 5 6 7 8 9)`.

On doit cependant retarder ce retournement autant que possible !

Deux autres exemples :

Exemple 1.

```scheme
; liste des entiers de a à b-1 (liste à l'envers)
(define (range2 a b)
  (if (= a b) '()
    (cons (- b 1) (range2 a (- b 1)))))

; liste des entiers de a à b-1 (liste à l'endroit)
(define (range2* a b) (reverse (range2 a b)))
```

Exemple 2.

```scheme
; conversion d'un entier strictement positif en binaire
; les chiffres sont rangés par poids croissants
(define (binaire n)
  (if (zero? n) '()
    (cons (modulo n 2) (binaire (quotient n 2)))))
```

Variables de portée limitée

Il est possible de définir des variables de portée limitée :

```scheme
; définition d'une variable x
(let ((x 5))
  (* x x))

; définition de deux variables a et b
(let ((a 10) (b 5))
  (+ a b))
```

Dans l'exemple suivant, on utilise la même variable deux fois :

```scheme
(define (powers2 k)
  (if (zero? k) '(1)
    (let ((p (powers2 (- k 1))))
      (cons (* 2 (car p)) p))))
```

Ici, la réutilisation permet d'économiser un appel récursif !

Exemple de la suite de Fibonacci

Analyser le code suivant :

```
(define (fibonacci n)
  (cond
    ((zero? n) '(0))
    ((= 1 n) '(1 0))
    (else
      (let ((f (fibonacci (- n 1))))
        (cons (+ (car f) (cadr f)) f)))))
        ; (cadr f) est l'abbréviation de (car (cdr f))
```

→ utilisation de `cond` pour effectuer plusieurs tests

→ rôle crucial du `let` (nombre d'appels récursifs)

→ construction typique de la liste à l'envers

→ intérêt de l'ordre inversé (accès aux derniers éléments)

Exercices d'application

→ compter les occurrences d'une valeur dans une liste
→ accéder au k-ième élément d'une liste n^{th}
→ renvoyer l'indice de la première occurrence d'une valeur
→ renvoyer la liste des indices des occurrences d'une valeur
→ vérifier que tous les éléments d'une liste sont non nuls
→ rechercher le minimum d'une liste
→ calculer la moyenne d'une liste de nombres
→ décoder un nombre binaire (liste à l'envers des chiffres)
→ valider un nombre binaire (liste à l'envers des chiffres) :
  – vérifier que la liste ne contient que des 1 et des 0
  – vérifier que le dernier chiffre est un 1
→ convertir un nombre dans une base arbitraire