Université Nice Sophia Antipolis

Les listes (chaînées)

Chap. 8

Les données structurées

- Jusqu'à présent, nous avons principalement travaillé sur des données *atomiques* (insécables) comme les nombres ou les booléens.
- Seules les images étaient des données composées [d'autres images !]. Mais il n'était pas possible de *déconstruire* une image… 
- La **structuration des données** va nous permettre d'envisager une valeur comme étant composée de plusieurs autres valeurs et de pouvoir accéder à ces valeurs. Nous avons déjà vu les **structures** dans le cours 2.
- En maths, l'exemple typique est un produit cartésien A x B x C dont les éléments sont les triplets (a,b,c) avec a ∈ A, b ∈ B, c ∈ C. Et l'exemple typique de produit cartésien est l'espace vectoriel $\mathbb{R}^n$.
- Nous allons nous focaliser en Scheme sur les **listes**, qui représentent les suites finies de valeurs L = (a b ...). Mais attention, **ce ne sont pas les listes de Python**, ce sont des **listes chaînées**.

Le type liste de Scheme

- Une **liste** est une suite finie de valeurs.

Expression	*Résultat*	
`(define L (list 6 4 5 8 3))`	*void*	*Définition d'une liste en extension*
`(list 6 4 5 8 3)`	`(6 4 5 8 3)`	*Construction d'une liste en extension*
`L`	`(6 4 5 8 3)`	*Affichage d'une liste*
`(first L)`	`6`	*Accès au premier élément*
`(rest L)`	`(4 5 8 3)`	*Accès à la liste privée du premier élément*
`(list? L)`	`true`	*Reconnaisseur de listes*
`(length L)`	`5`	*Nombre d'éléments*
`(empty? L)`	`false`	*La liste est-elle vide ?*
`(cons 0 L)`	`(0 6 4 5 8 3)`	*Construction d'une nouvelle liste par ajout d'un élément en tête*

N.B. La fonction `(cons x L)` *ne modifie pas la liste L. C'est bien une fonction* cons *: Elément x Liste → Liste, qui construit une nouvelle liste dont le premier élément est x et dont le reste est la liste L.* ***On rajoute à gauche, pas à droite !***

- La primitive `(list x1 x2 ...)` est une fonction, donc elle évalue ses arguments avant de construire la liste des valeurs obtenues :

```
> (define L (list (* 2 3) (+ 4 5)))
> L
(6 9)
```

```
> (first L)
6
```

```
> (rest L)
(9)
```

- Il est important de savoir utiliser la **quote** afin de distinguer une demande de calcul et une donnée brute sous forme de liste…

```
> (define L (+ 1 2 3))
> L
6
```

une demande de calcul (appel de fonction)

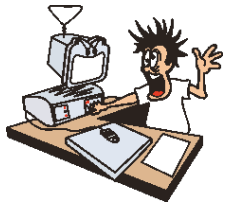
```
> (define L '(+ 1 2 3))
> L
(+ 1 2 3)
```

une donnée à l'état brut (ne pas calculer !)

```
> 'bonjour
bonjour
```

Première application : des fonctions à plusieurs résultats !

• Exemple dans les entiers naturels : comment programmer par récurrence la **division de a par b** si elle n'existait pas ? Elle doit retourner **deux résultats** : le quotient q et le reste r, sous la forme d'une **liste** `(q r)`.



- Le quotient de a par b, c'est 1 de plus que le quotient de a-b par b.
- Le reste de la division de a par b est le même que celui de la division de a-b par b.
- Donc a décroît. Cas de base lorsque a < b.

POUR diviser a par b et calculer (q,r) :

- si a < b, facile : le résultat est (0,a).
- sinon, je suppose par HR que je sais calculer la division (q_1,r_1) de a-b par b. Mais alors, la division de a par b n'est autre que $(q,r) = (q_1+1,r_1)$.


```
(define (division a b)        ; a et b ∈ N, b > 0, retourne le couple (q,r)
  (if (< a b)
    (list 0 a)
    (local [(define HR (division (- a b) b))]     ; HR = Hyp. de Récurrence
      (list (+ 1 (first HR)) (second HR)))))
```

```
> (define d (division 19 5))
> d
(3 4)
> (printf "La division de 19 par 5 s'écrit 19=5*~a+~a\n"
          (first d) (second d))
La division de 19 par 5 s'écrit 19=5*3+4
```

• Pour une fonction à trois résultats, on retournerait une liste à trois éléments, etc. Bien voir que l'Hypothèse de Récurrence produit une liste !

first, second, third, fourth....

• Premier aide-mémoire sur les listes :

		Complexité	
`list`	*Obj* × ... × *Obj* → *Liste*	$\mathcal{O}(n)$	} *en nombre d'appels à cons*
`cons`	*Obj* × *Liste* → *Liste**	$\mathcal{O}(1)$	
`first`	*Liste** → *Obj*	$\mathcal{O}(1)$	
`second`	*Liste*** → *Obj*	$\mathcal{O}(1)$	
`rest`	*Liste** → *Liste*	$\mathcal{O}(1)$	`L[1:]` *est* $\mathcal{O}(n)$ *en Python*
`empty?`	*Obj* → *Booléen*	$\mathcal{O}(1)$	

• **Principe de récurrence sur les listes** [TRES IMPORTANT !] :

Pour programmer par récurrence une fonction (foo L) portant sur une liste L :
- je commence par examiner le cas de la liste vide.
- si la liste est ≠ Ø, je suppose que je sais calculer (foo (rest L))
et je montre comment je peux en déduire la valeur de (foo L).

Quelques fonctions primitives sur les listes

• Nous allons programmer les principales fonctions Scheme prédéfinies sur les listes. Il faudra bien comprendre à la fois leur fonctionnement et leur complexité pour savoir les utiliser dans les algorithmes !

`(append L1 ... Lk)`	$\mathcal{O}(n_1 + \cdots + n_{k-1})$
`(build-list n f)`	$\mathcal{O}(n)$
`(length L)`	$\mathcal{O}(n)$
`(member x L)`	$\mathcal{O}(n)$
`(reverse L)`	$\mathcal{O}(n)$
`(sort L p)`	$\mathcal{O}(n \log n)$

• Rappel : dans ce cours, la complexité $\mathcal{O}(n)$ dénote un nombre d'opérations proportionnel à n dans le pire des cas ! Vous aurez des définitions mathématiques plus précises en maths...

La **longueur** d'une liste : (length L)

- La **longueur** d'une liste est le nombre de ses éléments en surface :

```
(length '(6 4 #t (5 a 1) 8 "une chaîne" coucou))  ⟶  7
```

- Programmation par ***récurrence sur (la longueur de) L*** :

  - si L est vide : sa longueur est 0, c'est le cas de base.

  - sinon, supposons par HR que l'on sache calculer la longueur de (rest L). Quid de la longueur de L ? Facile, c'est 1 de plus...

```
(define ($length L)   ; $ pour ne pas tuer la primitive length !
  (if (empty? L)
      0
      (+ 1 ($length (rest L)))))
```

```
> ($length '(a b (c d e) f g h))
6
```

avec un coût de 6...

COMPLEXITE DU PARCOURS en **O(n)** : Le nombre d'éléments de L si l'on mesure le nombre d'appels à rest.

L'**appartenance** à une liste : (member x L)

- La fonction **(member x L)** retourne #t ou #f suivant que x est ou non un élément *de surface* de la liste L.

- Exemples :

```
> (member 'qui '(le chien qui est noir))
#t
> (member 'qui '(le chien (qui est noir) mange vite)
#f
```

qui *n'est pas en surface...*

- Programmation :

```
(define ($member x L)
  (cond ((empty? L) #f)
        ((equal? x (first L)) #t)
        (else ($member x (rest L)))))
```

```
(define ($member x L)
  (and (not (empty? L))
       (or (equal? (first L) x)
           ($member x (rest L)))))
```

COMPLEXITE DU PARCOURS :
O(n) si l'on mesure le nombre d'appels à rest.

L'accès à **l'élément numéro k** d'une liste : (list-ref L k)

- Les éléments sont numérotés à partir de 0, comme dans tous les langages de programmation. Par exemple (first L) ⇔ (list-ref L 0).

```
> (list-ref '(jaune rouge bleu noir) 2)
bleu
```

- Voici comment est implémenté list-ref en Scheme :

```
(define ($list-ref L k)      ; 0 ≤ k < length(L)
  (cond ((empty? L) (error "$list-ref : Liste trop courte"))
        ((= k 0) (first L))
        (else ($list-ref (rest L) (- k 1)))))
```

- La **complexité** [nombre d'appels à rest] est clairement en **O(k)**. Ce n'est donc pas du tout la même chose qu'en Python, où le calcul de len(L) se fait en O(1). Les listes Scheme sont des *listes chaînées* alors que les listes Python sont des *tableaux*.

- MORALE : on s'efforcera de ne PAS utiliser list-ref en Scheme ! Contentez-vous d'avancer dans une liste par récurrence...

Un constructeur de liste en **compréhension** : (build-list n f)

- Intéressant si l'on connaît la **loi de formation du terme numéro i** :

```
(build-list 5 sqr)                  ⟶  (0 1 4 9 16)
(build-list 5 (lambda (i) (* 2 i)))  ⟶  (0 2 4 6 8)
```

la fonction qui exprime comment se calcule l'élément numéro i. Attention, les numéros commencent à 0.

COMPLEXITE DE LA CONSTRUCTION en **O(n)** si l'on mesure le nombre d'appels à cons.

```
(define ($build-list n f)
  (if (= n 0)
      empty       ; la liste vide !
      (cons (f 0) ($build-list (- n 1) (lambda (i) (f (+ i 1)))))))
```

(f(0) | f(1) f(2) f(3) ...)

La **concaténation** de deux listes : (append L1 L2)

- La fonction **(append L1 L2)** retourne une liste contenant les éléments de L1 juxtaposés à ceux de L2 :

```
> (append '(le chien que) '(je vois est noir))
(le chien que je vois est noir)
```

- Récurrence sur L1 : (le chien que) ⊕ (je vois est noir) = (le chien que je vois est noir)

```
(define ($append L1 L2)
  (if (empty? L1)
      L2
      (cons (first L1) ($append (rest L1) L2))))
```

- COMPLEXITE. On fait autant d'appels à cons que d'éléments dans L1. Donc le coût est **$O(n_1)$** et indépendant de L2 !

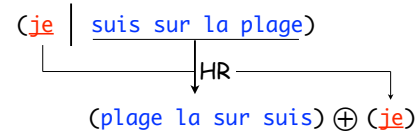
- Généralisation : (append L_1 L_2 ... L_k), COMPLEXITE : **$O(n_1+...+n_{k-1})$**

L'inversion d'une liste : (reverse L)

- Cette fonction (reverse L) retourne une copie inversée de L :

```
> (reverse '(je suis sur la plage))
(plage la sur suis je)
```

- Récurrence sur L.
  Hypothèse de récurrence :
  je sais inverser le reste de L !




```
(define ($reverse L)          ; algorithme naïf en O(n²)
  (if (empty? L)
      L
      (append ($reverse (rest L)) (list (first L)))))
```

- COMPLEXITE. Soit c_n le coût en nombre d'appels à cons pour inverser une liste de longueur n. Alors $c_0 = 0$ et $c_n = c_{n-1} + 1 + (n-1) = c_{n-1} + n$, d'où $c_n = O(n^2)$. Nous verrons plus tard un meilleur algorithme en **O(n)**...

Pourquoi donc $c_n = O(n^2)$?

- Comment raisonner sur une telle équation $c_n = c_{n-1} + n$?
- Si l'on possède des théorèmes : $c_n = c_{n-1} + O(n^d) \Rightarrow c_n = O(n^{d+1})$.
- Sinon, courage, on déplie la formule jusqu'à *voir* une loi de formation :

$c_n = c_{n-1} + n$
$c_n = c_{n-2} + (n-1) + n$
$c_n = c_{n-3} + (n-2) + (n-1) + n$
..... *; je pousse jusqu'au bout !*
$c_n = c_0 + (1 + 2 + ... + n)$ *; Ah-ah ! Formule connue...*
$c_n = c_0 + n(n+1)/2$
$c_n = O(n^2)$

L'algorithme est quadratique !

10000 éléments	*20000 éléments*
843 ms	*3275 ms*

Savoir si une liste est triée

- Une liste (x0 x1 x2 ...) est **triée** [*en croissant*] si $x_i \leq x_{i+1} \quad \forall i$
- Comment **savoir si une liste est triée ?** En la parcourant et en cherchant s'il existe une *inversion* (x_i, x_{i+1}) : telle que $x_i > x_{i+1}$.
- Hypothèse de récurrence : je sais si le reste de la liste est trié. Il me suffit alors de comparer les deux premiers éléments :

```
(define (croissante? L)
  (cond ((empty? L) #t)
        ((empty? (rest L)) #t)      ; un seul élément ?
        ((<= (first L) (second L)) (croissante? (rest L)))
        (else #f)))
```

```
> (croissante? '(2 8 8 12 23))
#t
> (croissante? '(2 6 8 12 10 23))
#f
```

COMPLEXITE : **O(n)** si l'on mesure le nombre d'appels à rest.

Le tri primitif d'une liste : (sort L rel)

```
> (sort '(12 6 2 23 8) <)
(2 6 8 12 23)
```

- Problème plus difficile mais fondamental. Il existe plusieurs manières de le résoudre ! Les meilleurs algorithmes ont une COMPLEXITE [nombre d'appels à cons] en **O(n log n)**.
- Par exemple la primitive Racket `(sort L rel)` où `rel` est une relation d'ordre strict quelconque [par exemple <].

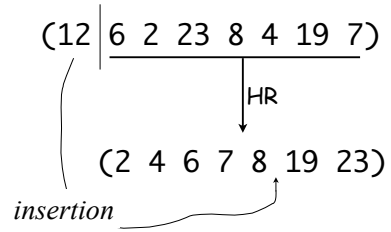
```
> (define L (build-list 20 (λ (i) (random 100))))
> L
(70 46 81 77 92 57 81 24 62 51 46 59 97 94 25 0 69 85 69 63)
> (sort L <)
(0 24 25 46 46 51 57 59 62 63 69 69 70 77 81 81 85 92 94 97)
```

- Vérifions que le tri est rapide [en n log n] :

```
> (define L1 (build-list 10000 (λ (i) (random 100))))
> (time (void (sort L1 <)))      ; void pour ne pas voir le résultat
cpu time: 2 real time: 3 gc time: 0          ; 2 millisecondes
> (define L2 (build-list 100000 (λ (i) (random 100))))
> (time (void (sort L2 <)))
cpu time: 23 real time: 23 gc time: 0        ; 23 millisecondes
```


Un algorithme de tri naïf : le TRI PAR INSERTION

- Voici la manière la plus simple de trier une liste L de nombres. On procède par récurrence brutale sur L.
- **Hypothèse de récurrence : je sais trier le reste de L !**




```
(define (tri-ins L)
  (if (empty? L)
      L
      (insertion (first L) (tri-ins (rest L)))))
```

- Il reste à programmer la fonction d'insertion !

- Soit donc à définir la fonction `(insertion x LT)` qui construit une nouvelle liste obtenue en insérant x à sa place dans la liste **triée** LT.
- Récurrence sur LT : supposons qu'on sache insérer x dans le reste de la liste LT. Comment en déduire l'insertion de x dans LT ?

```
(define (insertion x LT)
  (cond ((empty? LT) (list x))          ; (list x) ⇔ (cons x empty)
        ((< x (first LT)) (cons x LT))
        (else (cons (first LT) (insertion x (rest LT))))))
```

```
> (insertion 8 '(4 7 9 10))
(4 7 8 9 10)
> (tri-ins '(12 6 2 23 8 4 19 7))
(2 4 6 7 8 12 19 23)
```

- COMPLEXITE. Soit c_n le coût [nombre d'appels à cons] de trier une liste de longueur n. Alors $c_0 = 0$ et $c_n = c_{n-1} + $ *<coût de l'insertion>*.
Or le coût d'une insertion dans une liste de longueur n est en O(n).
Donc $c_n = c_{n-1} + O(n)$. Il en resulte que $c_n = \mathbf{O(n^2)}$. Pas fameux...

Rendre le tri polymorphe

- Et si je souhaite un tri décroissant, dois-je programmer un autre algorithme de tri ? Non, il me suffit de **faire abstraction de la relation d'ordre strict** < et la passer en paramètre :

```
(define (tri-ins L rel?)              ; rel? : L x L → boolean
  (if (empty? L)
      L
      (insertion (first L) (tri-ins (rest L) rel?) rel?)))

(define (insertion x LT rel?)         ; rel? est une relation d'ordre
  (cond ((empty? LT) (list x))
        ((rel? x (first LT)) (cons x LT))
        (else (cons (first LT) (insertion x (rest LT) rel?)))))
```

```
> (tri-ins '(12 6 2 23 8) <)          > (tri-ins '(12 6 2 23 8) >)
(2 6 8 12 23)                         (23 12 8 6 2)
> (tri-ins '("laetitia" "rachid" "kevin" "brice") string<?)
("brice" "kevin" "laetitia" "rachid")
```

(tri-ins (...) (lambda (x1 x2) ...))