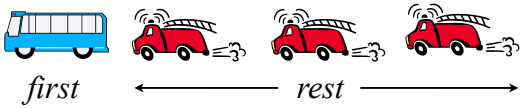
Université Nice Sophia Antipolis

Les listes (suite)

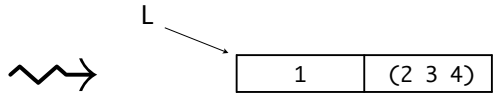


Chap. 8

Les listes "chaînées" de Scheme/Lisp

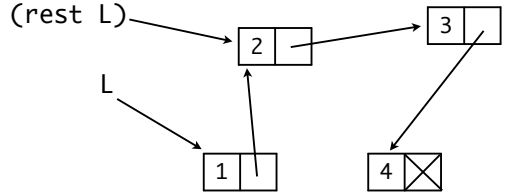
• Le mot **liste** recouvre deux structures de données distinctes suivant les langages de programmation. Commençons par celles de Scheme qui sont des **chaînages de doublets**.

```
> L
(1 2 3 4)
```

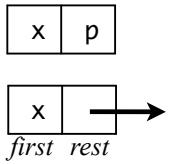


• RAPPEL : Une liste est *vide* ou bien constituée :
- d'un premier élément, accessible par la fonction `first`
- et du reste de la liste, accessible par la fonction `rest`

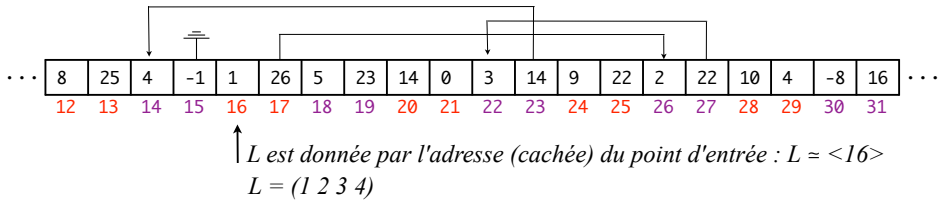
```
> (rest L)
(2 3 4)
```



• Donc au fond, une liste non vide est un **couple** <x,p> formée du premier élément x et d'un *pointeur* p vers le reste de la liste. Ce pointeur représente une adresse mémoire. Un tel couple se nomme un **doublet**.



• Les doublets sont éparpillés dans la mémoire des listes. Chaque doublet connaît le doublet suivant (pas le précédent !)...



DEFINITION : Une **liste** est définie par récurrence :
- ou bien c'est la constante liste vide `empty` notée aussi `'()`
- ou bien c'est un doublet dont le second élément (le reste) est une liste.

• Très bien, mais comment construire des doublets ?


```
(define L (list 1 2 3 4))
```

⇕

```
(define L (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
```

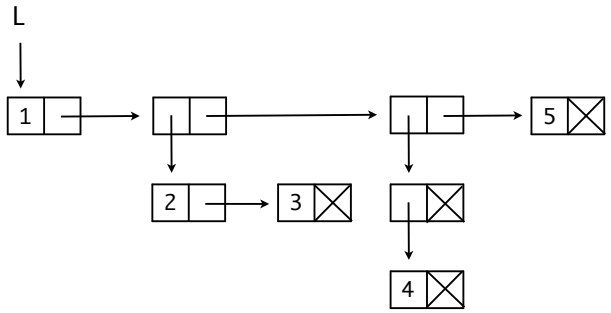
(1 2 3 4) ⟵ (2 3 4) ⟵ (3 4) ⟵ (4) ⟵ ()

• L'unité d'occupation mémoire pour les listes est le **doublet**. La liste L contient 4 doublets (voir page 2). *The memory footprint of L is 4 pairs !*

• La complexité du tri par insertion était de $O(n^2)$ doublets. Il s'agit du nombre de doublets construits durant l'exécution du tri, et non pas le nombre de doublets du résultat. La plupart de ces doublets ne serviront à rien ensuite et seront recyclés automatiquement par le **Garbage Collector** (GC). Tous ces doublets rendus à la mémoire libre seront chaînés et placés dans une **liste libre**, dans laquelle cons va puiser !

• Les architectures de doublets peuvent être *ramifiées* : une liste peut contenir d'autres listes !

```
> (define L (list 1 (list 2 3) (list (list 4)) 5))
> L
(1 (2 3) ((4)) 5)
```

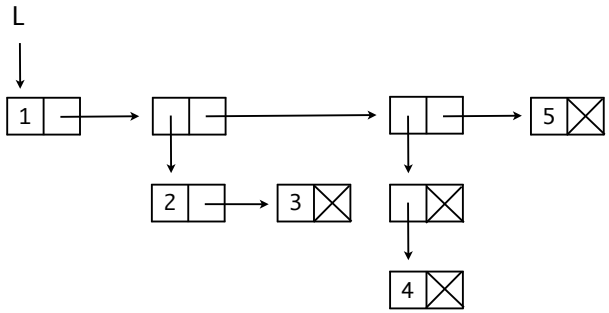


8 doublets
⇕
8 cons

```
(define L (cons 1 (cons (cons 2 (cons 3 empty))
                        (cons (cons (cons 4 empty) empty)
                              (cons 5 empty)))))
```


• Correspondance entre le dessin et la représentation parenthésée :

- une **flèche verticale** + une boîte ⇝ une parenthèse ouvrante.
- un **élément dans le FIRST** ⇝ on affiche le FIRST
- une **flèche horizontale** + une boîte ⇝ un espace.
- une **croix dans un REST** ⇝ on affiche une parenthèse fermante et on remonte.



(1 ◡ (2 ◡ 3) ◡ ((4)) ◡ 5)

Les listes de Python sont des "tableaux"

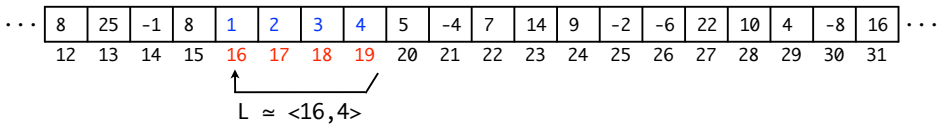
• Point de vue très différent. Une liste Python n'est pas chaînée, mais vue comme une suite de valeurs ***contigües en mémoire***.

```
> L
[1, 2, 3, 4]
```

```
> L[2]
3
```
O(1)

```
> len(L)
4
```
O(1)

```
> L[1:]
[2, 3, 4]
```
O(n)



• L est donnée par l'adresse de son point d'entrée 16 et sa longueur 4.

• L'accès à la longueur et au k-ème élément est immédiat, en O(1).

• MAIS pas de fonction `rest` ! Il faut recopier la tranche L[17:20]. Coûteux dans un raisonnement par récurrence...

• Le modèle réel est celui du type `list` de Python. Voir :

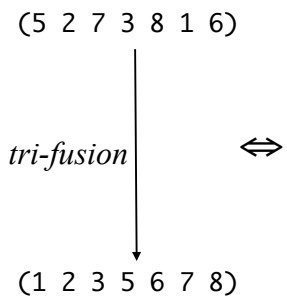
http://www.laurentluce.com/posts/python-list-implementation/

Le tri d'une liste par fusion

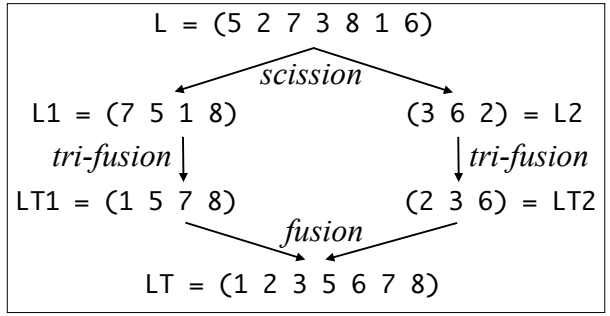
• Dans le cours 6 nous avions vu le *tri par insertion*, quadratique.

• Nous allons étudier le **tri par fusion** d'une liste L, qui procède par **dichotomie**, et sera donc sans doute plus rapide :

- je commence par scinder la liste L en deux sous-listes L1 et L2 de même longueur, ou presque.
- je trie récursivement L1 et L2, pour obtenir LT1 et LT2.
- je fusionne LT1 et LT2 en une seule liste triée.



⇔



- Au final, il y a trois fonctions à programmer :

`(tri-fusion L)`	☞	`LT`
`(scission L)`	☞	`(LT1 LT2)`
`(fusion LT1 LT2)`	☞	`LT`

- Voici la scission, vous rédigerez les deux autres en TD/TP. Le raisonnement - comme d'habitude - se fait par récurrence sur L. Je vais avancer de ***deux éléments à la fois*** à chaque étape, en plaçant le premier dans LT1 et le second dans LT2. S'il n'en reste qu'un, je le mettrai dans LT1 :

HR : Supposons que l'on connaisse `(scission (rest (rest L)))`.

```
(define (scission L)
  (cond ((empty? L) (list empty empty))
        ((empty? (rest L)) (list L empty))
        (else (local [(define HR (scission (rest (rest L))))]
                (list (cons (first L) (first HR))
                      (cons (second L) (second HR)))))))
```



```
> (scission '(5 2 7 3 8 1 6))
((5 7 8 6) (2 3 1))
```

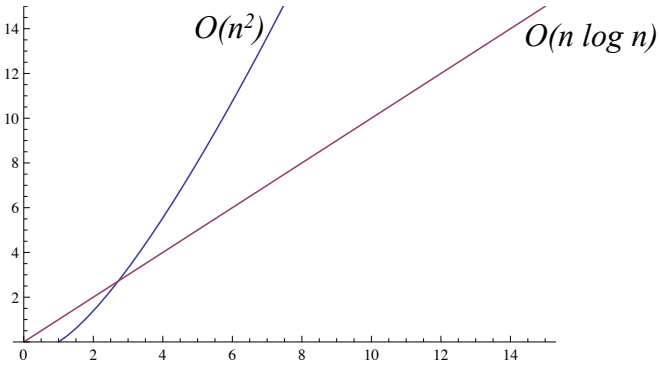
OK...

- En nombre d'appels à cons, le coût de `(scission L)` est clairement en O(n) puisqu'on parcourt linéairement la liste en demandant le même nombre de cons chaque fois. *Tiens, quel est le coût précis ???*
- Vous produirez aussi en TD/TP des algorithmes linéaires - en O(n) - pour la fonction `fusion`. Comment calculer la complexité de `tri-fusion` ?
- Soit c_n le coût [en nombre d'appels à cons] de tri-fusion d'une liste de longueur n. L'algorithme récursif de la page 8 fournit l'équation :

c_n = *<coût de scission>* + *<coût des HR>* + *<coût de fusion>*
$c_n = O(n) + 2\,c_{n/2} + O(n)$
$c_n = 2\,c_{n/2} + O(n)$
$c_n = 2\,c_{n/2} + n$ *pour simplifier !*

N'hésitons pas à simplifier le raisonnement, nous faisons des mathématiques d'ingénieur :-)

- Comment diable résoudre cette ***équation récursive*** $c_n = 2\,c_{n/2} + n$?
- Vous le verrez en TD, et vous obtiendrez $c_n = O(n \log n)$.
- L'algorithme obtenu est quasi-linéaire, juste un peu moins efficace que O(n), mais vraiment très peu... Beaucoup plus rapide que $O(n^2)$!
- MORALE : La stratégie ***couper en deux*** est encore gagnante.



Exemples issus de l'arithmétique et de la cinématique

Un entier est-il premier ?

- n entier est **premier** si n ≥ 2 et si son plus petit (et seul) diviseur dans [2,n] est précisément n.
- Calculons donc `(ppdiv n)`, le plus petit diviseur ≥ 2 de n.
- La récurrence brutale ne marche pas, je ne sais pas déduire `(ppdiv n)` à partir de `(ppdiv (- n 1))`. Je dois <u>généraliser le problème</u>.
- Cherchons le plus petit diviseur ≥ k de n, soit `(ppdiv>= n k)`.

```
(define (ppdiv>= n k)
  ; le plus petit diviseur de n qui soit ≥ k
  .....)
```

cf TP !

```
> (premier? 1)
#f
> (premier? 2)
#t
> (premier? 2011)
#t
> (premier? 2013)
#f
```

- Alors je sais tester si un nombre est premier :

```
(define (premier? n)
  (and (>= n 2) (= (ppdiv>= n 2) n)))
```


- L'algorithme précédent s'effondre sur les grands entiers, mais on sait faire mieux avec plus de maths (PCPS exercice 6.8.13)...
- Théorème d'Euclide : **il existe une infinité de nombres premiers**.

PREUVE. Supposons que cet énoncé soit faux. Il existerait alors un plus grand nombre premier N. Posons P = N! + 1. Puisque P > N, P ne peut pas être premier, il est donc divisible par un nombre premier q et nécessairement q ≤ N. Mézalor q serait à la fois un diviseur de N! et un diviseur de P = N! + 1, donc il diviserait leur différence 1, ce qui est impossible. L'hypothèse qu'il existe un plus grand nombre premier est donc absurde, CQED.

- La distribution des nombres premiers reste mystérieuse. Ils se raréfient à l'infini (on peut montrer qu'il y a des trous aussi grands qu'on veut, par exemple 100000000000 entiers consécutifs tous non premiers). Jusqu'à n, il y en a environ $n/ln(n)$, démontré en 1896.
- Le plus grand nombre premier connu (janvier 2013) est $2^{57885161}-1$

Liste de nombres premiers. Version 1.

- Calculons la liste `(premiers n)` des nombres premiers de [2, n]. Première tentative, récurrence brutale.

```
(define (premiers n)        ; les nombres premiers de [2,n]
  (if (<= n 1)
      empty
      (local [(define HR (premiers (- n 1)))]
        (if (premier? n) (cons n HR) HR))))
```

```
> (premiers 80)
(79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2)
> (length (premiers 10000))      ; time = 8.5 sec
1229
> (round (/ 10000 (log 10000)))
#i1086.0
```

- La liste est rendue à l'envers. Rustine : on l'inverse. Mais attention !!!

- Attention à inverser seulement à la fin :

```
(define (premiers n)                          ; dans l'ordre !
  (local [(define (aux n)                     ; produit la liste à l'envers
            (if (<= n 1)
                empty
                (local [(define HR (aux (- n 1)))]
                  (if (premier? n) (cons n HR) HR))))]
    (reverse (aux n))))
```

ajout à gauche

```
> (premiers 80)
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79)
```

- Et surtout résister à la tentation d'ajouter n à droite de HR par :

```
(if (premier? n) (append HR (list n)) HR)
```

car le coût serait $O(1) + O(2) + ... + O(n) = O(n^2)$, catastrophe ! On payerait 1000000 au lieu de 1000.

• Bon, c'est bien joli, mais peut-on obtenir la liste en ordre croissant **sans** payer `reverse` à la fin ? Réfléchissons :

• On obtient un mauvais ordre en descendant de n à n-1.
Idée : suffirait-il de monter au lieu de descendre ?...

• Je ne peux pas passer de n à n+1, je filerais vers l'infini !

• Mais je peux prendre une seconde variable k, comme à la page 13. Une variable qui va monter ! Cela revient encore à généraliser le problème :

Calculons la liste `(premiers>= n k)` des nombres premiers de [k, n].

```
(define (premiers>= n k)   ; les nombres premiers de [k,n]
  (cond ((> k n) empty)
        ((premier? k) (cons k (premiers>= n (+ k 1))))
        (else (premiers>= n (+ k 1)))))
```

```
(define (premiers n)   ; les nombres premiers de [2,n]
  (premiers>= n 2))    ; et zou ! On l'obtient dans l'ordre...
```


Liste de nombres premiers. Version 2.

• Par le CRIBLE D'ERATOSTHENE.

1. On part de la liste des entiers de [2,n] :

```
L = (2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... n)
```

2. Le premier élément candidat est premier, je le garde et je supprime ses multiples. Je passe à l'élément suivant et continue ainsi jusqu'à ce que tous les éléments aient été traités.

• Exemple :

```
(2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 ...)
(2 3 5 7 9 11 13 15 17 19 21 23 25)
(2 3 5 7 11 13 17 19 23 25)
(2 3 5 7 11 13 17 19 23)
(2 3 5 7 11 13 17 19 23)
```

etc.

cf TP !

Animations : et si le Monde était une liste ?

• Intéressons-nous aux **systèmes de particules**. Une *particule* est dotée de certaines propriétés, notamment sa position *(x,y)* et sa vitesse *(dx,dy)*. La vitesse *(dx,dy)* est vue comme une petite variation de la position. A chaque top d'horloge, la particule passe de la position *(x,y)* à la position *(x+dx, y+dy)*.

• La particule se déplace en général dans un **champ de gravitation**. Elle sera soumise à l'accélération verticale (on prendra g = 1) de la pesanteur, force constante tirant la composante *dy* vers le bas : à chaque top d'horloge, *dy* devient *dy+1* alors que la composante *dx* reste constante.

• **Le Monde sera une liste L de particules**. Le Monde initial INIT sera une liste de 20 particules sortant du centre du canvas, vitesse aléatoire vers le haut (en volcan). Une particule sortante renaît au centre du canvas, donnant l'illusion d'un flot continu !


```
(define (artifice)
  (local [(define FOND (place-image (rectangle 12 150 'solid "white")
                                    150 230 (rectangle 300 300 'solid "black")))
          (define BILLE (circle 6 'solid "yellow"))
          (define-struct bille (x y dx dy))   ; position et vitesse
          (define (random-bille)   ; qui nait au centre du canvas
            (make-bille 150 150 (- (random 11) 5) (- (random 20))))
          (local [(match-define (bille x y dx dy) (first L))]
          (define (suivant L)   ; Monde --> Monde
            (if (empty? L)
                L
                (local [(define b (first L)) (define x (bille-x b)) (define y (bille-y b))
                        (define dx (bille-dx b)) (define dy (bille-dy b))]
                  (cons (if (> y 300)
                            (random-bille)     ; renaissance !
                            (make-bille (+ x dx) (+ y dy) dx (+ dy 1)))   ; gravitation = 1
                        (suivant (rest L))))))
          (define (dessiner L)
            (if (empty? L)
                FOND
                (local [(define b (first L))]
                  (place-image BILLE (bille-x b) (bille-y b) (dessiner (rest L))))))]
    (big-bang INIT
              (on-tick suivant)
              (on-draw dessiner))))
```

```
(define (artifice)
  (local [(define FOND (place-image (rectangle 12 150 'solid "white")
                                    150 230 (rectangle 300 300 'solid "black")))
          (define BILLE (circle 6 'solid "yellow"))
          (define-struct bille (x y dx dy))   ; position et vitesse
          (define (random-bille)   ; qui nait au centre du canvas
            (make-bille 150 150 (- (random 11) 5) (- (random 20))))
          ; Le Monde est une liste de billes
          (define INIT (build-list 20 (lambda (i) (random-bille))))
          (define (suivant L)   ; Monde --> Monde
            (if (empty? L)
                L
                (local [(match-define (bille x y dx dy) (first L))]
                  (cons (if (> y 300)
                            (random-bille)     ; renaissance !
                            (make-bille (+ x dx) (+ y dy) dx (+ dy 1)))   ; gravitation = 1
                        (suivant (rest L))))))
          (define (dessiner L)
            (if (empty? L)
                FOND
                (local [(match-define (bille x y dx dy) (first L))]
                  (place-image BILLE x y (dessiner (rest L))))))]
    (big-bang INIT
              (on-tick suivant)
              (on-draw dessiner))))
```

Version finale !

Lisez le document animation.pdf !