

Université de Pau et des Pays de l'Adour

Le langage Scheme

eric.gouarderes@univ-pau.fr

<https://webcampus.univ-pau.fr/courses/PROGFONC/>



Langage fonctionnel

- ◆ Ecrire un programme : définir une fonction

$$f : x, y \longrightarrow x + y$$

- ◆ Exécuter un programme : appliquer une fonction

$$(f \ 2 \ 5)$$

- ◆ Récursivité

$$\begin{array}{c} \mathbb{N} \times \mathbb{N} \\ f : x, n \longrightarrow x^n \end{array}$$

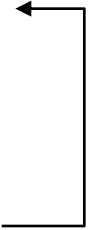
$$\left\{ \begin{array}{ll} 1 & n=0 \\ x \cdot x^{n-1} & n>0 \end{array} \right.$$

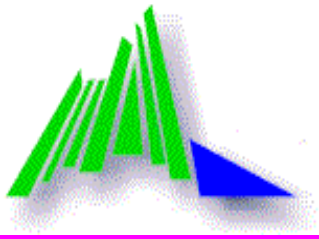


Langage Scheme

- ◆ Langage fonctionnel de la famille LISP (Steele et Sussman , MIT 1975)
 - traitement de listes
 - expressions symboliques

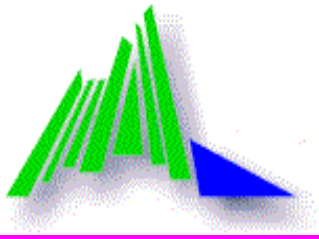
 - ◆ Langage interprété
 - **Evaluateur : cycle R.E.P.L.**
comparable au fonctionnement
d 'une calculette
- Read**
Eval
Print
Listen


-
- ◆ Syntaxe des expressions basée sur une notation préfixée totalement parenthésée



Quelques références

- **La programmation, une approche fonctionnelle et récursive avec Scheme** - Laurent Ardit, Stéphane Ducasse - Eyrolles, 1996
- **Débuter la programmation avec Scheme** - Jean-christophe Routier et Eric Wegrzynowski -International Thomson Publishing, 1997
- **Programmer avec Scheme, de la pratique à la théorie** -Jacques Chazarain - International Thomson Publishing, 1996
- **L'environnement de programmation DRScheme** :
<http://www.drscheme.org/>



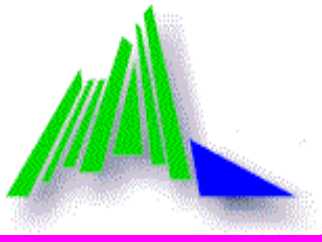
Les Différentes expressions Scheme

◆ simple

- nombres : 5, 6.5, 1/3
- les valeurs booléennes : #t, #f
- chaînes : « bonjour », « au revoir »

◆ symbole :

- représente
 - un identifiant d'une expression (variable, fonction)
 - une valeur symbolique
- suite de caractères alphanumériques sans espace ni accent. Le premier caractère est toujours un caractère alphabétique ou symbolique : x, a21, +, -, and, if...



Les Différentes expressions Scheme

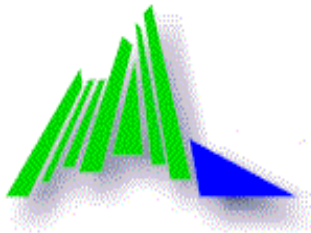
◆ liste :

- représente
 - l'application d'une fonction : $5 + 6 \rightarrow (+\ 5\ 6)$
 - un ensemble de données : $(a, b, c) \rightarrow (a\ b\ c)$
- chaque élément de la liste est une expression. Chaque élément est séparé des autres par un *espace*

◆ lambda expression

- représente une fonction (dite fonction anonyme)
- liste de trois éléments commençant par le mot clé **lambda**
(*lambda liste_paramètres_formels corps_de_la_fonction*)
les paramètres formels et le corps de la fonction sont des expressions Scheme

☞ En scheme, une fonction est une expression de 1^o classe

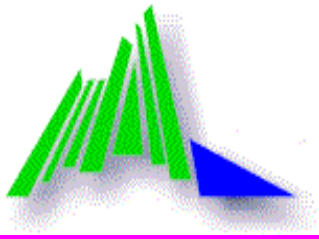


Notion d 'environnement

- ◆ Zone mémoire dans laquelle sont stockées toutes les définitions de variables et de fonctions. On peut la voir comme un tableau d'associations entre un symbole (nom de variable ou de fonction) et une valeur (expression scheme).

Symbole	Valeur
a	10
pi	3.1415
addition	(lambda (x y) (+ x y))

- ◆ Utilisé pour l 'évaluation des expressions
 - ☞ au départ contient tous les symboles prédéfinis
 - ☞ structure hiérarchique




Evaluation des expressions

◆ Les Lambda expressions

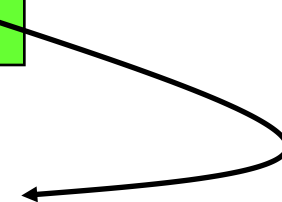
- le résultat de l'évaluation d'une lambda est **une fermeture** (closure). Structure précisant les paramètres formels, le corps et l'environnement d'évaluation

évaluation de (lambda (x y) (+ x y))

Args :	Corps :	Env. :
(x y)	(+ x y)	

**Environnement
d'évaluation**

[liaisons initiales]





Exemples d'évaluation d'expression

Expression

Résultat d'évaluation

- -12

→ -12

- #f

→ #f

- « bonjour »

→ « bonjour »

- (+ 2 3)

→ 5

- (+ (- 3 1) (+ 2 3) 4 (* 3 7)) → 32

- (and (> 12 45) (= 3 (/ 12 4))) → #f

- (or #t (< 12 3))

→ #t



Quelques fonctions arithmétiques

n-aire

- ◆ $(+ 2 3 5 3) \rightarrow 13$
- ◆ $(* 2 3 5) \rightarrow 30$
- ◆ $(- 2 3 4) \rightarrow -5$
- ◆ $(/ 2 3 3) \rightarrow 0.2222222222222222$

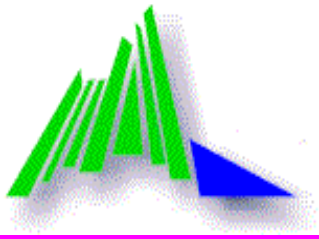
binaire

- ◆ $(\text{quotient } 5 2) \rightarrow 2$
- ◆ $(\text{remainder } 5 2) \rightarrow 1$
- ◆ $(\text{modulo } 5 2) \rightarrow 1$
- ◆ $(\text{expt } 2 3) \rightarrow 8$

- | | |
|----------------------------|------------------|
| $(\text{remainder } -5 2)$ | $\rightarrow -1$ |
| $(\text{modulo } -5 2)$ | $\rightarrow 1$ |
| $(\text{remainder } 5 -2)$ | $\rightarrow 1$ |
| $(\text{modulo } 5 -2)$ | $\rightarrow -1$ |

unaire

- ◆ $(\text{abs } -3) \rightarrow 3$
- ◆ $(\text{sqrt } 9) \rightarrow 3$



Fonctions booléennes

◆ not : négation, fonction unaire

- (not #t) \rightarrow #f
- (not #f) \rightarrow #t

◆ and : et logique, fonction n-aire

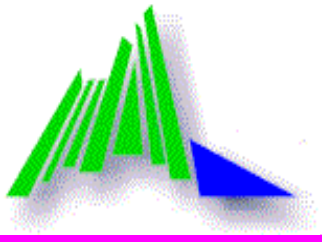
- (and ...#f ...) \rightarrow #f
- (and #t #t #t) \rightarrow #t

◆ or : ou logique, fonction n-aire

- (or ... #t ...) \rightarrow #t
- (or #f #f #f) \rightarrow #f

☞ (or (= 2 2) (= 3 (/ 5 0))) \rightarrow #t, sans erreur

☞ (and (= 2 3) (= 3 (/ 5 0))) \rightarrow #f, sans erreur

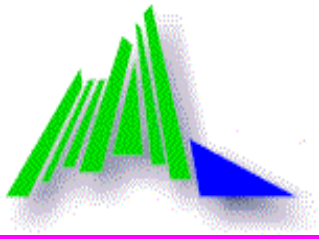


Fonctions relationnelles

- ◆ Fonctions n-aires
- ◆ Résultat booléen (#t ou #f)
- ◆ =, <, >, <=, >=

$(= 2\ 3\ 4) \rightarrow \text{\#f}$

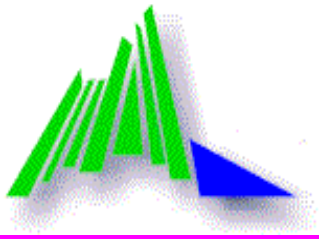
$(\leq x\ 5) \left\{ \begin{array}{ll} \rightarrow \text{\#t} & \text{si } x \text{ est inférieur ou égal à } 5 \\ \rightarrow \text{\#f} & \text{si } x \text{ est supérieur à } 5 \end{array} \right.$



Fonctions prédicats

- ◆ Résultat Booléen (#t, #f)
- ◆ Le nom se termine par ?
- ◆ Equal ?
 - fonction binaire
 - teste l'égalité de deux expressions (pas seulement des nombres)

(equal? x « a ») $\left\{ \begin{array}{ll} \rightarrow \text{\#t} & \text{si } x \text{ vaut « a »} \\ \rightarrow \text{\#f} & \text{si } x \text{ différent de « a »} \end{array} \right.$



Fonctions prédicats

◆ zero?

- unaire
- teste si une expression est nulle

(zero? x) $\left\{ \begin{array}{ll} \rightarrow \text{\#t} & \text{si } x \text{ vaut } 0 \\ \rightarrow \text{\#f} & \text{si } x \text{ différent de } 0 \end{array} \right.$

◆ boolean?, number?, symbol?, string?, integer?, real?...

- unaire
- teste le type d'une expression

(integer? x) $\left\{ \begin{array}{ll} \rightarrow \text{\#t} & \text{si } x \text{ est un nombre entier} \\ \rightarrow \text{\#f} & \text{si } x \text{ n'est pas un nombre entier} \end{array} \right.$



La fonction define - définition de variables et de fonctions

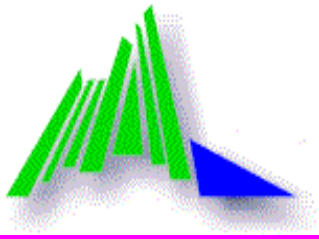
(**define** *symbole* *expression*)

- ◆ fonction binaire
- ◆ L 'application de la fonction define
 - n'évalue pas le 1^o argument, évalue le second (**fonction spéciale**)
 - A pour effet d 'ajouter l'association à l 'environnement
 - donne **une valeur et un type d 'expression** au symbole
 - moyen pour définir **des variables et des fonctions**

(define x 10)



Symbole	Valeur
x	10



La fonction define - définition de variables et de fonctions

(**define** *symbole* *expression*)

- ◆ fonction binaire
- ◆ L 'application de la fonction define
 - n'évalue pas le 1^o argument, évalue le second (**fonction spéciale**)
 - A pour effet d 'ajouter l'association à l 'environnement
 - donne **une valeur et un type d 'expression** au symbole
 - moyen pour définir **des variables et des fonctions**

	Symbole	Valeur
(define x 10)	x	10
(define y (+ x 5))	y	15



La fonction define - définition de variables et de fonctions

(**define** *symbole* *expression*)

- ◆ fonction binaire
- ◆ L 'application de la fonction define
 - n'évalue pas le 1^o argument, évalue le second (**fonction spéciale**)
 - A pour effet d 'ajouter l'association à l 'environnement
 - donne **une valeur et un type d 'expression** au symbole
 - moyen pour définir **des variables et des fonctions**

	Symbole	Valeur			
(define x 10)	x	10			
(define y (+ x 5))	y	15			
(define add (lambda (x y) (+ x y)))	add	<table><tr><td>(x y)</td><td>(+ x y)</td><td></td></tr></table>	(x y)	(+ x y)	
(x y)	(+ x y)				



La fonction define - définition de variables et de fonctions

(**define** *symbole* *expression*)

- ◆ fonction binaire
- ◆ L 'application de la fonction define

Ne retourne pas de résultat !

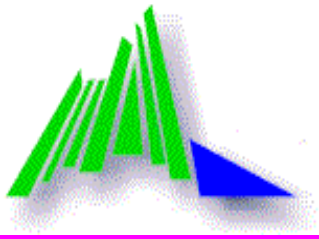
(define x 10) → **Indefini**

(define y (+ x 5)) → **Indefini**

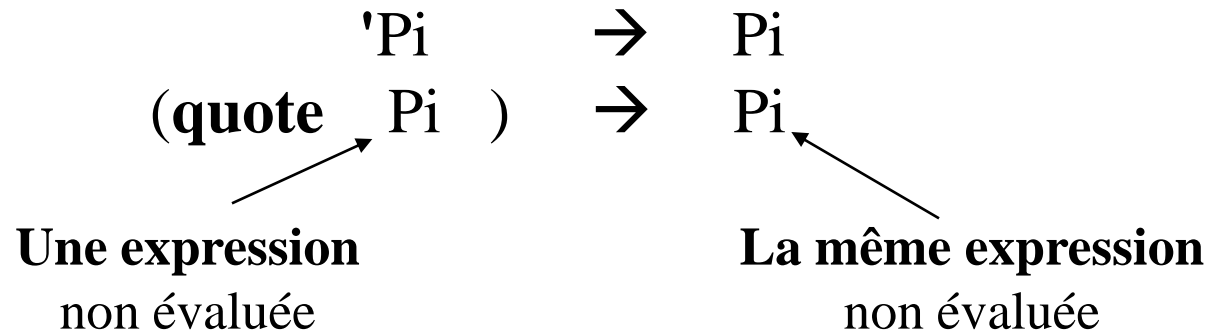
(define add

Symbole	Valeur			
x	10			
y	15			
add	<table><tr><td>(x y)</td><td>(+ x y)</td><td></td></tr></table>	(x y)	(+ x y)	
(x y)	(+ x y)			

(lambda (x y) (+ x y))) → **Indefini**

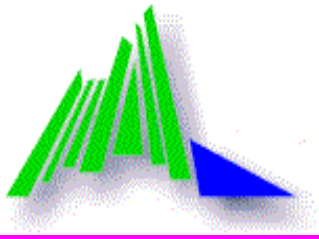


Le mécanisme de citation



- ◆ L 'application de la fonction quote retourne l 'expression non évaluée

- ☞ permet de manipuler des valeurs symboliques
- ☞ permet d 'utiliser les listes comme structure de donnée



Le mécanisme de citation

- ◆ (define voiture R19)

- ☞ problème car R19 doit être évalué

- ☞ symbole non défini

- ◆ (define voiture 'R19)

- ☞ crée la liaison: voiture \longrightarrow R19

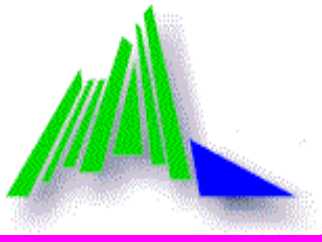
- ☞ voiture variable symbolique de valeur R19

- ◆ (define 1 (+ 5 6))

- ☞ crée la liaison: 1 \longrightarrow 11

- ◆ (define 1 '(+ 5 6))

- ☞ crée la liaison : 1 \longrightarrow (+ 5 6)



Séquence et affectation

◆ (begin <suite d 'expressions>)

- évaluation en séquence des expressions
- retourne le résultat de la dernière évaluation
- Exemple : (begin (+ 5 1) (* 6 5)) --> 30

◆ (set! symbole expression)

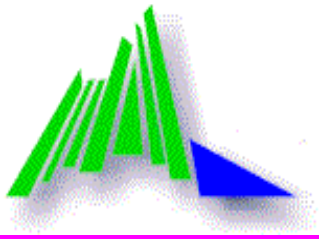
- evalue *expression* et affecte le résultat à symbole
- ne retourne rien
- attention *symbole* doit être défini avant
- Exemple

☞ (define x 10) -->

☞ (set! x (+ x 1)) -->

☞ x --> 11

☞ Ne pas abuser de l 'affectation



Fonction conditionnelle If

- ◆ (if *expr_test* *expr_alors* *expr_sinon*)
 - fonction spéciale :
évalue soit *expr_alors* soit *expr_sinon* en fonction de *expr_test*
 - Exemple :
(if (x < 0)
 ‘ négatif
 ‘ positif_ou_nul)
 - Forme simplifiée : (if *expr_test* *expr_alors*)
- ☞ Attention : *expr_alors* et *expr_sinon* représentent un expression unique. Pour évaluer plusieurs expressions utiliser begin



Fonction conditionnelle Cond

◆ (**cond**

(expr_test_1 <suite d 'expressions_1>)

(expr_test_2 <suite d 'expressions_2>)

...

(expr_test_n <suite d 'expressions_n>)

(**else** <suite d 'expressions>)

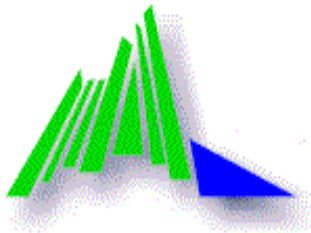
)

- Généralisation de la fonction IF

- fonction spéciale :

- 1) évalue successivement chaque test jusqu'à trouver un test vrai ou trouver le mot clé **else**

- 2) Evalue la suite d 'expressions correspondantes et retourne le résultat de la dernière expression évaluée.



Fonction conditionnelle Cond

(cond

```
((> 3 5) (+ 100 3))  
((> 10 1) (+ 100 4))  
((< 0 2) (+ 100 5))  
(else 88)) → 104
```

(define (signe x)

(cond

```
((< x 0) « le nombre est négatif »)  
((= x 0) « le nombre est nul »)  
(else « le nombre est positif))
```

)

(define couleur (animal)

(cond

```
((equal? animal 'elephant) '(couleur : gris))  
((equal? animal 'zebre) '(couleur : blanc_raye_noir))  
((equal? animal 'chien) '(couleur : je ne sais pas))  
(else '( je ne connais pas cet animal)))
```

)



Notion d 'environnement temporaire (environnement local)

- ◆ **(let** ((*variable₁* *expression₁*)
 (*variable₂* *expression₂*)
 ...
 (*variable_n* *expression_n*)
)
 expression
)
- ◆ Fonction binaire
- ◆ Définition et utilisation de variables temporaires (locales)
- ◆ Le premier argument est une **liste de couples** (en bleu).
Un couple associe un symbole (variable) à une expression.
- ◆ Le second argument est le **corps** de la fonction let. C 'est
une expression (en rose) .



Notion d 'environnement temporaire (environnement local)

◆ **(let** ((*variable*₁ *expression*₁)
 (*variable*₂ *expression*₂)
 ...
 (*variable*_{*n*} *expression*_{*n*})
)
 expression
)

Environnement *initial* (*global*)

Environnement <i>Et</i>	
Symbole	Valeur
variable₁	eval_expression₁
variable₂	eval_expression₂

...

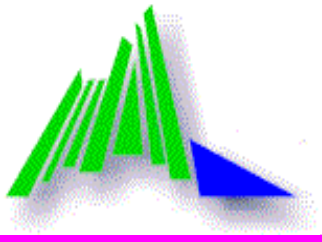
...

Variable_{<i>n</i>}	eval_expression_{<i>n</i>}
------------------------------------	---

◆ Application d'une fonction let :

- 1) création d 'un environnement temporaire *Et*, lié (lien hiérarchique) à l 'environnement *initial*, appelé également environnement *global*,
- 2) pour chaque couple (*variable*_{*i*} *expression*_{*i*}) évaluation de *expression*_{*i*} et création de la liaison dans l 'environnement *Et*,
- 3) évaluation du corps *expression* dans l 'environnement *Et*,
- 4) suppression de l 'environnement *Et*.

☞ Les *expression*_{*i*} ne peuvent pas utiliser les variables du let



Notion d 'environnement temporaire (environnement local)

◆ (let ((*variable*₁ *expression*₁)
 (*variable*₂ *expression*₂)
 ...
 (*variable*_{*n*} *expression*_{*n*})
)
 expression
)

Environnement *initial* (*global*)

Environnement <i>Et</i>	
Symbole	Valeur
variable₁	eval_expression₁
variable₂	eval_expression₂

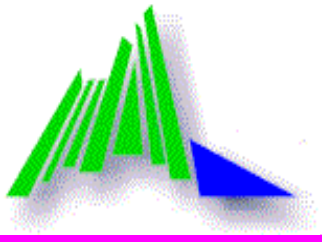
...

...

Variable_{<i>n</i>}	eval_expression_{<i>n</i>}
------------------------------------	---

◆ Si l' *expression* contient un ou plusieurs symboles, deux cas peuvent se produire lors de l'évaluation :

- le symbole est une des variables du let : l'évaluation retourne la valeur associée dans l'environnement *Et*,
- le symbole n'est pas une variable du let: l'évaluation retourne la valeur de l'environnement que l'on aurait trouvée sans le let (environnement initial).



Notion d 'environnement temporaire

Exemples

◆ (let ((x 5)
 (y 3)
)
 (* x y)
) → 15

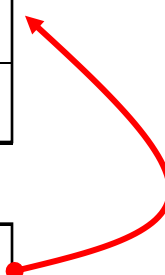
Environnement <i>initial</i>	
Environnement <i>Et</i>	
symbole	valeur
x	5
y	3



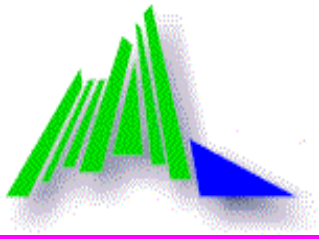
◆ (define x 100)
◆ (let ((x 2)
 (y (+ x 3))
)
 (* x y)
) → 206

Environnement <i>initial</i>	
symbole	valeur
x	100

Environnement <i>Et</i>	
symbole	valeur
x	2
y	103



Liens
hiérarchiques



Notion d 'environnement temporaire

Exemple de hiérarchie

```
◆ ( let ((x 100)
      )
    (let ((y 2)
          (z (+ x 3))
        )
      (* y z)
    )
  ) → 206
```

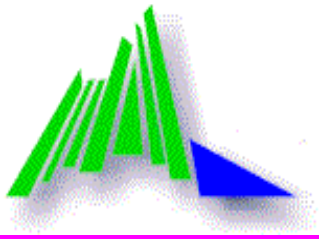
Environnement *initial*

Environnement <i>Et1</i>	
symbole	valeur
x	100

Environnement <i>Et2</i>	
symbole	valeur
y	2
z	103

Liens
hiérarchiques

Liens
hiérarchiques



Entrée/Sortie

◆ (Read)

- lecture d 'une expression , retourne l 'expression sans l 'évaluer
- Exemples

lecture sans mémorisation du résultat	lecture avec mémorisation dans une variable
<ul style="list-style-type: none">– (read) ; début de lecture– (+ 2 3) ; saisie de l 'expression– (+ 2 3) ; résultat	<ul style="list-style-type: none">– (define x (read)) ; debut de lecture– (+ 2 3) ; saisie de l 'expression– x --> (+ 2 3)

◆ (display expression)

- affiche le résultat de l 'évaluation de *expression*, **ne retourne rien**

◆ (newline)

- passage à la ligne, **ne retourne rien**



Entrée/Sortie

```
(define (f x)
```

```
  (cond ((number? x) (display x) (display « est un nombre » ))
```

```
        ((symbol? x) (display x) (display « est un symbole »))
```

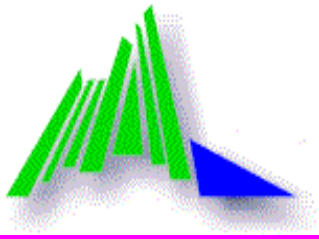
```
        (else (display x) (display « : je ne sais pas ce que c 'est »))))
```

(f 5) → indéfini

5 est un nombre

(f « bonjour ») → indéfini

bonjour : je ne sais pas ce que c 'est



Les listes

◆ Notion de couple (paire pointée)

- notation : **(a . b)**

Attention aux espaces !

- représentation : 

- Constructeur :

- **cons** (binaire) :

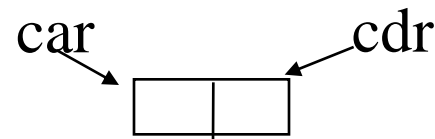
 **(cons 'a 'b) --> (a . b)**

 **(cons (+ 10 5) (cons 'a 'b)) --> (15 . (a . b))**

- Accès aux éléments

- **car** : **(car '(a . b)) --> a**

- **cdr** : **(cdr '(a . b)) --> b**



- Prédicat de type : **pair?**



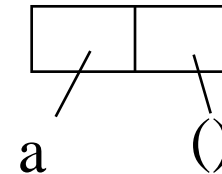
Les listes

◆ Définition

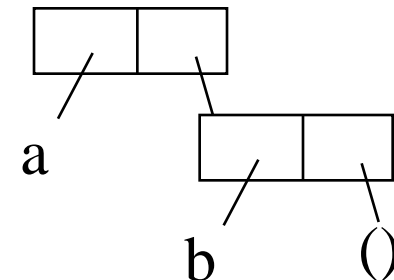
- la liste vide notée $()$
- une paire pointée dont le second élément est **une liste**

◆ Exemples

- $(a . ())$ liste à un seul élément



- $(a . (b . ()))$ liste à deux éléments



☞ pour chaque élément : une paire pointée



Les listes

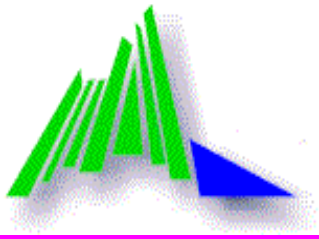
◆ Notation simplifiée

- (<suite d 'éléments séparés par des espaces>)
- (a. ()) --> (a)
- (a . (b . ())) --> (a b)

◆ Constructeurs :

- **cons** (binaire) : (cons 'a '(b c)) --> (a b c)
☞ attention le deuxième argument doit être une liste
- **list** (n-aire) : (list 'a 'b 'c) --> (a b c)
- **append** (n-aire) : (append '(a b) '(c d e)) --> (a b c d e)

☞ Remarque : utilisation du quote pour considérer la liste comme un ensemble de données.

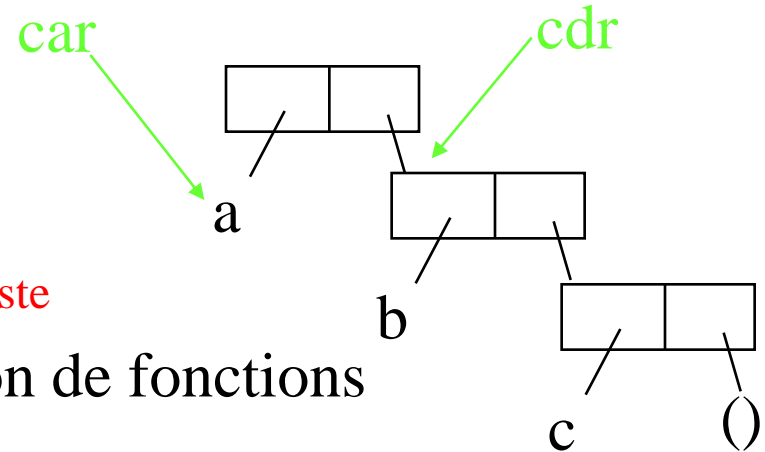


Les listes

◆ Accès aux éléments :

- `car : (car '(a b c)) --> a`
- `cdr : (cdr '(a b c)) --> (b c)`

☞ attention le `cdr` est une liste

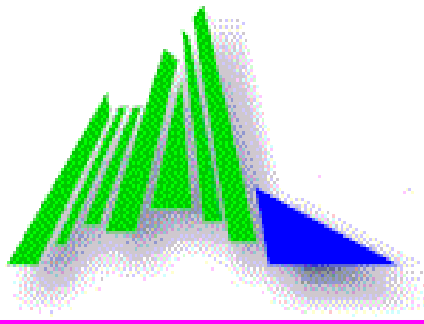


◆ Notation condensée : composition de fonctions

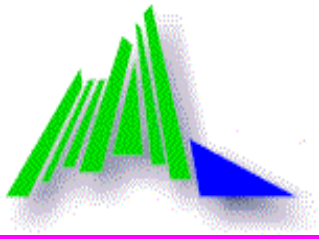
- `(car (car l)) ⇔ (caar l)`
- `(car (cdr l)) ⇔ (cadr l)`
- `(car (cdr (cdr (car l)))) ⇔ (caddar l)`

◆ Prédicats

- type : **list** ? `(list? '(a b c)) --> #t`
- Liste vide : **null**? `(null? '()) --> #t`
- Liste non vide : **pair**? `(pair? '(a)) --> #t`
`(pair? '()) --> #f`

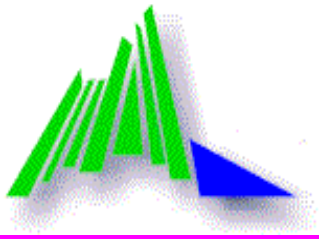


Récurtivité sur les listes



Principe

Une liste est une donnée **structurellement réursive** : elle est soit vide, soit composée d'un élément, **le car**, mis en tête d'un autre liste, **le cdr**. Pour cette raison, il est intéressant d'utiliser des fonctions récursives pour résoudre de nombreux problèmes sur les listes.



Exemple

- ◆ Ecrire la fonction **min** qui prend une liste de 3 nombres en argument et retourne le minimum.

Ex. : (min '(8 1 4)) → 1

```
(define (min l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2premier (if (< (car l) (cadr l))
                             (car l)
                             (cadr l))))
        (if (< (caddr l) min2premier)
            (caddr l)
            min2premier)))))
```

```
(define (min1 l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2dernier (if (< (cadr l) (caddr l))
                             (cadr l)
                             (caddr l))))
        (if (< (car l) min2dernier)
            (car l)
            min2dernier)))))
```



Exemple

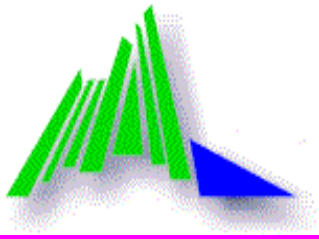
- ◆ Ecrire la fonction **min** qui prend une liste de 3 nombres en argument et retourne le minimum.

Ex. : (min '(8 1 4)) → 1

```
(define (min l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2premier (if (< (car l) (cadr l))
                            (car l)
                            (cadr l))))
        (if (< (caddr l) min2premier)
            (caddr l)
            min2premier)))))
```

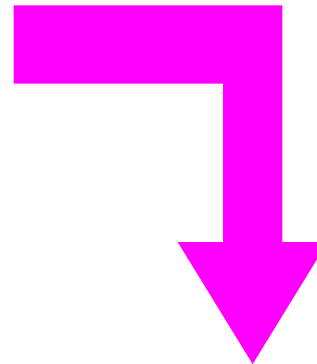
```
(define (min1 l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2dernier (if (< (cadr l) (caddr l))
                             (cadr l)
                             (caddr l))))
        (if (< (car l) min2dernier)
            (car l)
            min2dernier)))))
```

- ◆ Ecrire la même fonction pour une liste de longueur quelconque



Exemple

```
(define (min1 l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2dernier (if (< (cadr l) (caddr l))
                            (cadr l)
                            (caddr l))))
        (if (< (car l) min2dernier)
            (car l)
            min2dernier))))
```

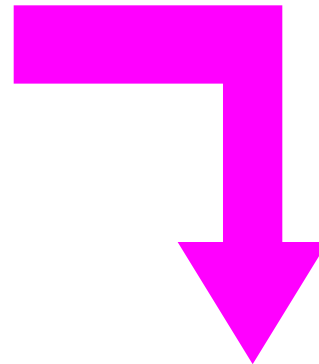


```
(define (min2 l)
  (cond ((null? l) 'Erreur_liste_vide)
        ((null? (cdr l)) (car l))
        (else (let ((minn-1derniers (min2 (cdr l))))
                  (if (< (car l) minn-1derniers)
                      (car l)
                      minn-1derniers))))))
```

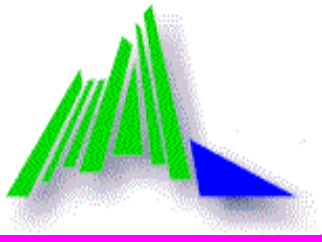


Exemple

```
(define (min l)
  (if (not (= (length l) 3))
      'erreur_nombre_elements
      (let ((min2premier (if (< (car l) (cadr l))
                            (car l)
                            (cadr l))))
        (if (< (caddr l) min2premier)
            (caddr l)
            min2premier))))
```



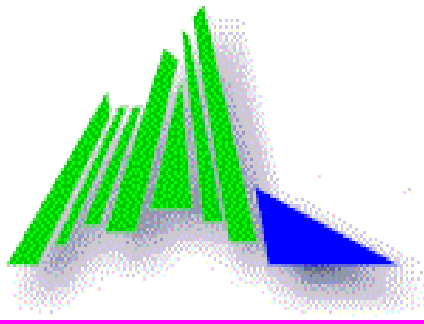
```
(define (min3 l)
  (cond ((null? l) 'Erreur_liste_vide)
        ((null? (cdr l)) (car l))
        ((< (car l) (cadr l))
         (min3 (cons (car l) (caddr l))))
        (else (min3 (cdr l)))))
```



Méthode

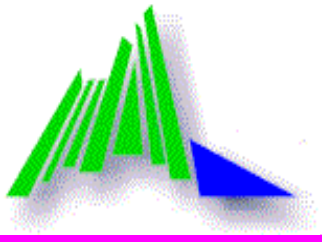
- ◆ La méthode d'écriture de ces fonctions est très souvent la même.
 - La variable de récursivité est la liste,
 - sa valeur minimale est (),
 - la fonction de variation est **cdr**: partant d'une liste non vide quelconque, puis en faisant un certain nombre de fois **cdr**, on arrive toujours à ().

- ◆ Quand on écrit une fonction prenant en paramètre une liste, il faut avoir le réflexe programmation récursive : **est-ce que je peux ramener le calcul de cette fonction au calcul de la même fonction sur le cdr de la liste?**



Université de Pau et des Pays de l'Adour

Quelques fonctions avancées



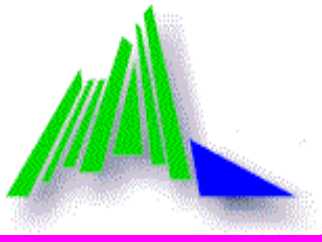
Fonction d'évaluation

◆ Evaluer une expression

(eval *expression*)

- $(+ 5 10) \rightarrow 15$
- $'(+ 5 10) \rightarrow (+ 5 10)$
- $(\text{eval } '(+ 5 10)) \rightarrow 15$

- $(\text{define } x \text{ ' } +)$
- $x \rightarrow +$
- $(+ 5 10) \rightarrow 15$
- $(x 5 10) \rightarrow \text{erreur}$
- $((\text{eval } x) 5 10) \rightarrow 15$



Fonction *apply*

- ◆ Applique une fonction à une liste d 'arguments
(**apply** *fonction* <liste>)

- ◆ Si $liste = (a_1 a_2 \dots a_n)$ et $fonction = f$

Equivaut à évaluer : $(f a_1 a_2 \dots a_n)$

- $(\text{apply } + '(5\ 4\ 6)) \rightarrow 15$
- $(\text{apply } \text{cons } '(a\ b)) \rightarrow (a . b)$



Fonction *map*

- ◆ Applique une fonction à chaque élément d'une liste
(**map** *fonction* <liste>)

- ◆ Si $liste = (a_1 a_2 \dots a_n)$ et $fonction = f$
Equivaut à évaluer : $(f a_1)$ et $(f a_2)$ et ... $(f a_n)$

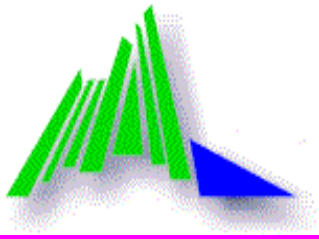
- ◆ Retourne la liste des évaluations : $((f a_1) (f a_2) \dots (f a_n))$
 - $(map\ number? '(5 \ll toto \gg x 5)) \rightarrow (\#t \#f \#f \#t)$
 - $(map\ cons '((a\ b\ c) (1\ 2\ 3))) \rightarrow ((a . 1) (b . 2) (c . 3))$



Macro caractères

◆ Quasi-citation et virgule

- ``s` = (quasiquote s)
- Se comporte presque comme quote : ``(a b) → (a b)`
- Différence concerne les listes avec possibilité d'évaluation sélective grâce au caractère ,
 - `(define x 1)`
 - `(define y '(u v))`
 - ``(x y ,x ,y) → (x y 1 (u v))`



Macro caractères

- ◆ Quasi-citation et couple virgule arobasque
 - (define x 1)
 - (define y '(u v))
 - `(x y ,x ,@y) → (x y 1 u v)
- L'expression qui suit ,@ est évaluée. Sa valeur doit être une liste et c'est le contenu de la liste qui est mis à sa place



Fonctions à arité variable

- ◆ Avec paramètres obligatoires
 - (lambda (x1 x2 ... xn . Lparam)
expression)
- ◆ Sans paramètres obligatoires
 - (lambda Lparam
Expression)

Exemple :

```
(define mcons (lambda (a . l) ; cons itéré 1 paramètre au moins
  (cons a
    (if (null? l)
        '()
        (apply mcons l)))))
```

(mcons 1 2 3 '(a b)) → (1 2 3 a b)