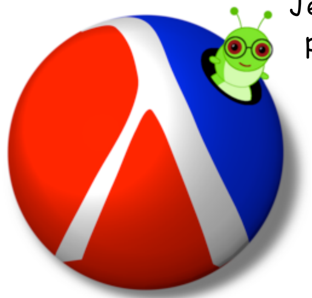
OPTION PF2 - Automne 2016
Jean-Paul Roy
L2-Info&Maths, Faculté des Sciences

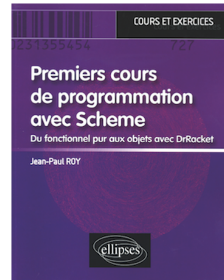
Cours n°1

http://deptinfo.unice.fr/~roy

Full Racket !

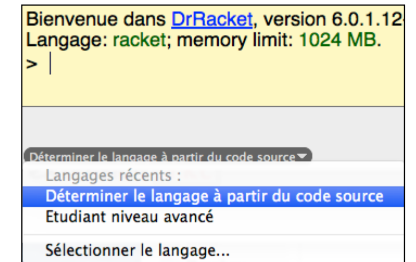


Je ne suis plus un débutant !



Chap. 11

① Les modules



- Dorénavant un programme Scheme sera un ensemble de **modules**. Comme en Python, avec ou sans des classes d'objets...

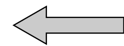
- Un fichier **module** :
  - **étend** un langage [par exemple `racket`] avec la directive `#lang`
  - **importe** d'autres modules avec la directive `require`
  - **exporte** variables, fonctions et macros avec la directive `provide`

- La **programmation modulaire** facilite grandement la réutilisabilité et le partage du code, et est particulièrement utile pour la réalisation de bibliothèques. Exemple : un type abstrait, ou une classe.

- Le nom d'un fichier module a pour extension `.rkt`

fichier "tva.rkt" ← *Un fichier utilitaire*

```
#lang racket
(provide TVA prix-tcc)    ; exportation
(define TVA 1.186)
(define (prix-tcc p)
  (* p TVA))
(printf "Module tva.rkt loaded !\n")
```



Mon programme

fichier "use-tva.rkt"

```
#lang racket
(require "tva.rkt")          ; importation (chemin relatif)
(define (bilan x)
  (printf "Avec un taux de ~a, le prix ttc est ~a\n"
          TVA (prix-tcc x)))
(bilan 26)
```


② Les doublets : une nouvelle vision de la fonction cons

- En PF1, la fonction `(cons x L)` prenait un élément `x` et une **liste** `L`. Elle construisait une nouvelle liste `R` telle que :

`(first R) = x   et   (rest R) = L`

- Exemple :

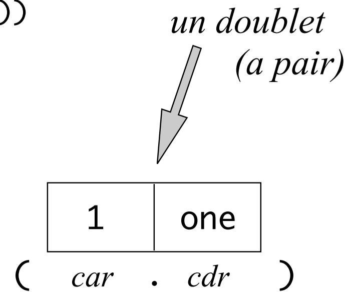
```
> (define R (cons 'ceci '(est bien connu)))
> R
(ceci est bien connu)
> (first R)
ceci
> (rest R)
(est bien connu)
```

- Donc la signature était `cons` : *Elément* x *Liste* → *Liste*

- Et la liste vide se notait `empty`.

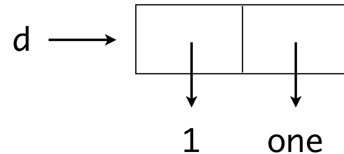
• En vrai Scheme comme en **Lisp**, la fonction `(cons x y)` peut prendre deux éléments de types quelconques, et retourne un **doublet** :

```
> (define d (cons 1 'one))
> d
(1 . one)
> (car d)
1
> (cdr d)
one
> (pair? d)
#t
```



• En réalité, d est un **pointeur** [une adresse mémoire] vers une zone comportant 2 mots mémoire de 32 ou 64 bits.

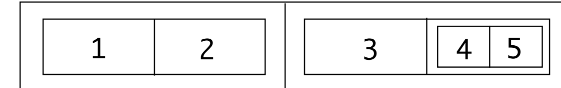
• Un doublet occupe donc toujours 8 octets en mémoire sur une machine 32 bits, quel que soit son contenu.



Chaînages de doublets

• Rien n'empêche d'emboîter les doublets :

```
> (define big (cons (cons 1 2) (cons 3 (cons 4 5))))
```



• Deux problèmes :

- le schéma devient vite difficile à lire. On va *sortir les boîtes...*
- l'affichage au toplevel **n'est pas** celui qu'on attend !

```
> big
((1 . 2) 3 4 . 5)
```

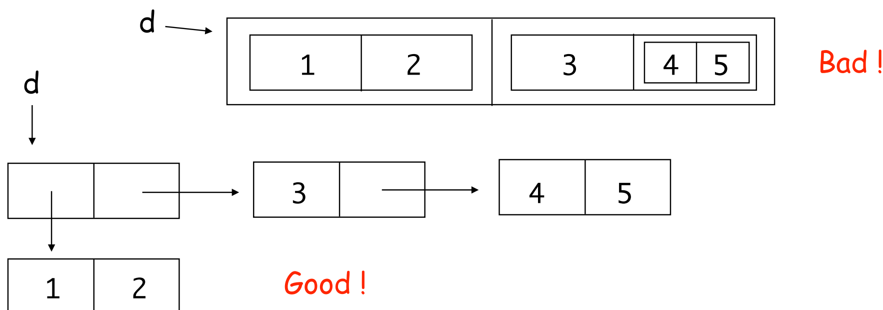
`((1 . 2) . (3 . (4 . 5)))`

1. Affichage simplifié : c'est la **convention du point-parenthèse** :

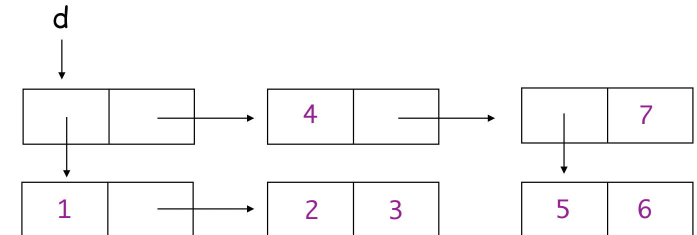
> *Lorsqu'un point est suivi d'une parenthèse ouvrante, on efface les deux, ainsi que la parenthèse fermante associée !*

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

2. On fait sortir les boîtes internes. Une boîte dans un CAR sort à la verticale, une boîte dans un CDR sort à l'horizontale (en principe) :



• Algorithme de production de ***l'affichage d'un chaînage de doublets*** :



- une **flèche verticale** + une boîte ⇝ une parenthèse ouvrante.
- un **élément dans un CAR** ⇝ on écrit le CAR.
- une **flèche horizontale** + une boîte ⇝ un espace.
- un **élément dans un CDR** ⇝ on affiche un point, le CDR, une parenthèse fermante et on remonte.

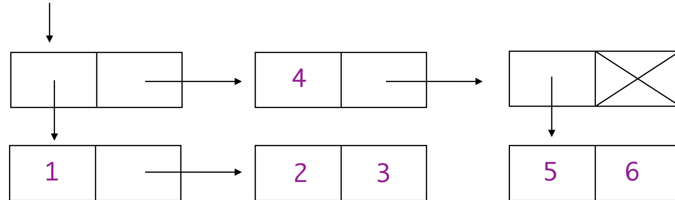
```
> (define d
    (cons (cons 1 (cons 2 3)) (cons 4 (cons (cons 5 6) 7))))
> d
((1 2 . 3) 4 (5 . 6) . 7)
```

Parcours en profondeur préfixe !

Les LISTES sont des chaînages particuliers de doublets

DEFINITION : Une **liste** est :
- soit la liste vide, notée empty ou '()
- soit un doublet dont le cdr est une **liste** (récurrence !).

• Exemple de liste : `L = ((1 2 . 3) 4 (5 . 6))`



• Il suffit de vérifier que le dernier CDR est une croix [liste vide] !

• Ou bien qu'il n'y a pas de *point* en avant-dernière position !

```
> (list? '(a (b . c) d))
#t
```

```
> (list? '(a b . d))
#f
```


③ La quote et la quasiquote

• La **quote** (accent aigu) existe pour empêcher l'évaluation prématurée d'une expression Scheme : `'(+ 1 2)` ⇔ `(quote (+ 1 2))`

• La **quasiquote** (ou **backquote** en Lisp, l'accent grave) semble identique à la quote. Mais à l'intérieur d'une quasiquote, on peut forcer l'évaluation d'une expression en la faisant précéder d'une virgule. Lorsque la virgule est suivie d'un @ et que la valeur de l'expression à évaluer est une liste, les éléments de cette liste sont directement incorporés dans la liste quasiquotée. Ouf.

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

```
> `(L vaut ,L)
(L vaut (1 2 3))
> `(et contient ,@L)
(et contient 1 2 3)
> `(START ,@(map sqr L) END)
(START 1 4 9 END)
```


④ Arguments optionnels *avec des paires pointées*

• Certaines fonctions Scheme sont d'**arité variable** :

```
> (+)
0
> (+ 1)
1
> (+ 1 2)
3
> (+ 1 2 3)
6
```

```
> (* 1 2 3 4 5 6 7 8 9 10)
6628800
> (max 7 2 4 8 5 3)
8
> (map (lambda (x y) (+ x (sqr y)))
       '(1 2 3) '(4 5 6))
(17 27 39)
etc.
```

• **Ce que le langage se permet, il DOIT vous le permettre !**

• Soit à programmer une fonction `(somcarrés x ...)` :

```
> (somcarrés 1 2 3 4 5)
55
```

```
> (somcarrés 1 2)
5
```

• Quel est le *point commun* entre les listes `(somcarrés 1 2 3 4 5)` et `(somcarrés 1 2)`, voire `(somcarrés)` tout court ?

• La forme générale d'un appel à la fonction somcarrés sera toujours de la forme `(somcarrés . L)` où L est la liste des arguments :

`(somcarrés 1 2 3 4 5)` ⇔ `(somcarrés . (1 2 3 4 5))` — où `(1 2 3 4 5)` est L

Version 1 : à l'***ordre supérieur***, en une ligne

```
(define (somcarrés . L)      ; L est la liste des arguments
  (apply + (map sqr L)))
```

Version 2 : avec une ***récurrence enveloppée***

```
(define (somcarrés . L)
  (if (empty? L)
      0
      (+ (sqr (car L)) (apply somcarrés (cdr L)))))
```

Version 3 : avec une ***itération***...

A faire...

⑤ Arguments optionnels individuels nommés

- La tendance (cf Python) est de **nommer** les arguments optionnels :

```
> (define (foo x (y 2) (z 3))
    (+ x y z))
> (foo 0)
5
> (foo 0 10)
13
> (foo 0 10 1000)
1010
```

```
>>> def foo(x,y=2,z=3) :
        return x+y+z
>>> foo(0)
5
>>> foo(0,10)
13
>>> foo(0,10,1000)
1010
```

- Ou bien à les passer par **mots-clés** (*keywords*) :

```
> (define (voiture n #:carburant c #:annee [a 2006])
    (list n c a))
> (voiture 'Suzuki #:carburant 'diesel)
(Suzuki diesel 2006)
>  (voiture 'suzuki #:annee 2010 #:carburant 'diesel)
(suzuki diesel 2010)
```

#:carburant *obligatoire !*

#:année *optionnel !*

⑥ Les variables locales

- L'unique manière de construire des **variables locales à un calcul** dans notre ancien langage PF1 *Etudiant avancé* était (local …).

```
(local [(define ...)
        ...]
  ...)
```

- Il s'agit d'une construction qui n'appartient pas au noyau Scheme, mais qui est **propre à Racket**, et bien pratique ! Vous pouvez donc continuer à l'utiliser. Vous la trouverez peu dans les livres (sauf PCPS). Nous allons donc détailler les constructions de ***variables locales normalisées de Scheme*** :

let   let*   define *interne*

- La construction fondamentale (letrec …) pour les fonctions récursives locales (cf L3-Info) est de moins en moins utilisée, et remplacée par une suite de define internes.

- La construction (let ((x₁ v₁) (x₂ v₂) ...) expr₁ expr₂ …) équivaut à la construction Python :

```
(x1, x2, ...) = (v1, v2, ...) ; expr1 ; expr2 ; ...
```

```
> (define x 2)
> (let ((x (+ x 1)) (y (* x 2)))
    (+ x y ))
7
> x
2
```

let

N.B. Ce n'est pas une construction essentielle du langage puisque'elle est équivalente à **l'application d'une fonction anonyme** (cf p. 18) :

```
((lambda (x y) (+ x y)) (+ x 1) (* x 2))
```

N.B. Racket moderne a tendance à snober let…

- La construction (let* ((x₁ v₁) (x₂ v₂) ...) expr₁ expr₂ …) équivaut à la construction Python :

```
x1 = v1; x2 = v2 ; ... ; expr1 ; expr2 ; ...
```

```
> (define x 2)
> (let* ((x (+ x 1)) (y (* x 2)))
    (+ x y ))
9
> x
2
```

let*

N.B. Ce n'est pas une construction essentielle du langage puisqu'elle est équivalente à une suite de let emboîtés :

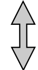
```
(let ((x (+ x 1)))
  (let ((y (* x 2)))
    (+ x y)))
```

N.B. Racket moderne préfère les define internes à let*...

⑦ Les définitions internes

• Il est possible de définir des ***sous-fonctions*** à l'intérieur des fonctions **sans utiliser** **local** :

```
(define (fac n)
  (local [(define (iter n acc)
            (if (= n 0)
                acc
                (iter (- n 1) (* n acc))))]
    (iter n 1)))
```



Interdit en "Etudiant avancé" (pourquoi ?)

Cool !

```
(define (fac n)
  (define (iter n acc)
    (if (= n 0)
        acc
        (* n (fac (- n 1)))))
  (iter n 1))
```

Un peu comme les def *internes de Python...*

⑧ Les MACROS, ou extensions syntaxiques

• Et si la forme (let ...) n'existait pas ? Ce ne peut PAS être une fonction puisque ses arguments ne sont pas évalués. C'est une **forme spéciale** ! Or :

```
(let ((x v) ...) expr ...)            (*)
```

```
((lambda (x ...) expr ...) v ...)     (**)
```

cf page 15

• L'équivalence ci-dessus est une **règle syntaxique** (*syntax rule*).

• Elle dit au compilateur : si tu vois une expression qui a la forme de l'expression (*), remplace-la par l'expression (**) !

• La primitive ***define-syntax*** permet au programmeur d'exprimer une telle équivalence syntaxique. Elle permet de **DEFINIR SOI-MEME UNE NOUVELLE FORME SPECIALE** comme and, or, let, while, for, etc.


```
(define-syntax mot-clé
  (syntax-rules (mots-réservés)
    (motif-initial motif-transformé)
    ...))
```

```
(define-syntax $let
  (syntax-rules ()                       ; aucun mot réservé
    (($let ((x v) ...) expr …)           ; ici une règle
     ((lambda (x ...) expr ...) v ...))))
```

• Autre exemple : un (si ... alors ... sinon ...)

```
(define-syntax si
  (syntax-rules (alors sinon)
    ((si p alors q sinon r) (if p q r))))
```

*N.B. Les **mots réservés*** alors *et* sinon *ne seront reconnus que dans le contexte d'une expression* (si ...).

MACRO = MOTEUR A DEUX TEMPS

```
(si (integer? pi) alors (* 2 3) sinon (+ 2 3))
```

⇓ ***expansion*** *du texte source*

```
(if (integer? pi) (* 2 3) (+ 2 3))
```

⇓ ***évaluation*** *du texte expansé*

5

MORALE : Une **macro** consiste à *bluffer le compilateur* et à lui faire absorber une syntaxe imprévue. Certains langages de programmation proposent l'écriture de macros. Mais Scheme possède les macros les plus abouties, dites **hygiéniques** (pas de conflit avec les noms des variables extérieures).

⑨ Mais où est passé check-expect ?…

• Il est réservé aux langages débutants. Il faudra en langage Racket importer le module **rackunit**.

```
#lang racket
(require rackunit)  ; aide en ligne sur l'API de RackUnit

(define (foo x)
  (if (number? x)
      (+ x 1)
      (error x "Should be a number !")))

(check-equal? (foo 5) 6)                        ; passe
(check-equal? (foo 3) 5)                        ; échoue
(check-exn exn:fail? (lambda () (foo 'a)))     ; passe
```

On attend une erreur…

```
FAILURE
actual:     4
expected:   5
name:       check-equal?
location:   (#<path:/Users/roy/Desktop/foo.rkt> 9 0 226 24)
expression: (check-equal? (foo 3) 5)
```


⑩ Et qu'en est-il de printf ?…

• Il est toujours là ! En PF2, nous pourrons l'incorporer dans une fonction en tant qu'**instruction** : **une expression sans résultat** [dont la valeur est #<void>, produite par un appel à (void)]. Par défaut, le toplevel n'affiche pas les valeurs #<void>.

```
> (printf "")
> (void? (printf ""))
#t
> (printf "~a\n" (printf "foo\n"))
foo
#<void>
```

```
> (void? (void))
#t
> (void? (+ 1 2))
#f
```

• Si vous programmez une fonction sans résultat, elle devra donc terminer en retournant (void).

Oui, mais comment l'incorporer dans une fonction ?

• En effet, nous avions dit en PF1 (langage *Etudiant avancé*) que le corps d'une fonction ne pouvait contenir qu'une seule expression :

```
(lambda (x ...) expr)
```

• En réalité il existe en Scheme (et en *Etudiant avancé* !) une forme spéciale (begin e1 e2 ...) permettant d'évaluer en séquence les expressions e1 e2 ... et retournant le résultat de la dernière ei.

```
(begin e1 e2 ...)
```

• En Python, elle s'écrit e1 ; e2 ; ... ou bien en les alignant à la verticale. En Scheme elle est **optionnelle** partout sauf dans le if.

```
(define (add x y)    ; pour pister les appels à +
  (printf "(add ~a ~a)\n")
  (+ x y))
```

begin *implicite…*

Itération : la panoplie des boucles for…

• Racket procure des boucles for, for/list, for/sum, etc. adaptées aux calculs **itératifs** un peu dans le style Python. Sauf qu'elles peuvent s'exploiter de manière fonctionnelle (cf TP).

```
> (for ([i (in-range 10 20 2)])         ;   for i in range(10,20,2) :
    (printf "~a " i))                   ;        print(i,end=' ')
10 12 14 16 18
> (for ([i (in-range 5)] [j '(a b c d e f)] #:when (odd? x))
    (printf "~a " (list i j)))
(1 b) (3 d)
> (define L (for/list ([i (in-range 10)] #:when (odd? i))
              (sqr i)))                 ;  une sorte de map-if…
> L
(1 9 25 49 81)
```

• Toutes ces boucles for sont bien entendu des macros Scheme qui génèrent des itérations usuelles.

Avec begin, peut-on modifier des variables ?

- Pas encore, manants ! Attendez le cours 2.

La Guerre de l'Affectation