Le langage Scheme : Fondamentaux

D'après "Programmer avec Scheme" de Jacques Chazarain, les cours de J.-P. Roy et de C. Queinnec

I. Primitives indispensables

Primitive dans les entiers	*Remarques*	*Résultats*
(modulo p q)	q > 0	p % q
(quotient p q)	p >= 0, q > 0	p / q
(gcd n_1 n_2 …)		PGCD
(lcm n_1 n_2 …)		LCM
(random n)	dans [0, n-1]	Nombre aléatoire
(zero? n)		n = 0 ?
(even? n)		pair ?
(odd? n)		impair ?

Primitive dans les rationnels	*Résultats*
(numerator r)	Numérateur de r
(denominator r)	Dénominateur de r
(rationalize x y)	Le rationnel le plus 'simple' ne différant de x qu'au plus de y

Primitive dans les réels	*Résultats*
(abs x)	| x |
(floor x)	Le plus grand entier <= x
(ceiling x)	Le plus petit entier >= x
(round x)	Arrondi
(min x_1 x_2 …)	Minimum
(max x_1 x_2 …)	Maximum
(random x)	x réel > 0, résultat réel dans [0, x[
(< x_1 x_2 …)	Test suite strict. Croissante
(<= x_1 x_2 …)	Test suite croissante
(> x_1 x_2 …)	Test suite strict. Décroiss.
(>= x_1 x_2 …)	Test suite décroissante

Primitive dans les complexes	*Remarque*
(+ z1 z2 …)	Somme
(* z1 z2 …)	Produit
(- z1 z2)	z1 – z2
(/ z1 z2)	z1 / z2
(/ z)	1 / z
(sqrt z)	Racine carrée principale de z
(exp z)	Exponentielle de base e
(exp z1 z2)	Exponentielle de base z1
(log z)	Le logarithme
(sin z), (cos z), (tan z), (asin z), (acos z), (atan z)	Trigonométrie classique
(= z1 z2 …)	Test de suite constante (danger si inexact !)

Primitive pour tout objet	*Remarques*
(number? obj)	Test nombre
(complex? obj)	Test complexe
(real? obj)	Test réel
(rational? obj)	Test rationnel
(integer? obj)	Test entier
(exact? obj)	Test si le nombre est exact, i.e. n'a pas de virgule.
(exact->inexact obj) (inexact->exact obj)	Conversion d'un nombre exact en inexact (rajout d'une virgule) ou l'inverse.
(string? obj)	Test chaîne de caractère
(procedure? obj)	Test fonction
(boolean? obj)	Test booléen

Remarques :
*(+) rend 0, car 0 est l'élément neutre pour l'addition. De même (*) rend 1, car 1 est l'élément neutre pour la multiplication. Mais (/) provoque simplement une erreur…*
equal? est pour des objets ; = pour nombres (equal regarde le type et dispatche les tâches)

Toute primitive valable dans un ensemble l'est également dans ses sous-ensembles. Du plus grand ensemble au plus petit, on a :

- objet
- complexe
- réel
- rationnel
- entier

Quelques lectures :
comp.lang.scheme *Programmer avec Scheme* (Jacques Chazarain)
Structure & Interpretation of Computer Programs (H. Abelson, G. Sussman : 1ère année du MIT)
Recueil de petits problèmes en Scheme (L. Moreau, C. Queinnec, D. Ribbens et M. Serrano)
The Scheme Programming Language (R. Kent Dybvig)

II. Fonctions[1] : define et lambda

Pour nommer une valeur par un identificateur, on dispose de la forme (define ident exp).
Exemple : (define pi 3.14159) (define rac2 (sqrt 2))
Dans un langage, il est important de pouvoir définir le plus de choses possibles à partir des primitives.
Ainsi, si on a besoin de définir les constantes e et pi :

```
(define e (exp 1))                    (define pi (acos -1)) ; cos pi = -1, on prend la réciproque
```

Si on veut nommer une expression contenant des paramètres, il s'agit d'une fonction.
On utilise à nouveau define : (define (nom-fonction x_1 x_2 … x_k)
corps de la fonction)

```
Exemples : (define (VolumeSphere x)        (define (hypotenus a b)                    (define (carre x)
             (* 4/3 pi x x x))                 (sqrt (+carre a) (carre b))))                (* x x))
```

Une fonction anonyme, ou fonction pure (du style $x \mapsto 2x + 1$) est donnée par lambda.
On a ainsi la forme (lambda (paramètres) (définition)).
Exemple : (lambda (x) (- (* 2 x) 1)) pour $x \mapsto 2x + 1$, ou (lambda (x y) (+ (* x x) (* y y))) pour $x^2 + y^2$.
Au top-level (la console) de DrScheme on peut les accompagner de valeurs : ((lambda (x) (x)) 3) donne 3.

Pour définir une fonction on a deux choix : montrer qu'une fonction est construire en évaluant une lambda (style puriste, ou Indiana), ou abstraire la forme d'appel (style M.I.T., plus général).
Style Indiana : (define carre (lambda (x) (* x x)))
Style MIT : (define (carre x) (* x x))

III. Conditions : if, cond et connecteurs logiques

Une condition 'si' s'écrit sous la forme (if condition conséquence alternant). Si la condition est vérifiée on exécute sa conséquence, et sinon on fait l'alternant.

```
Exemples : (define valeurAbsolue      ; x réel
             (lambda (x)
               (if (>= x 0) x (- x))))

           (define signe      ; x réel
             (lambda (x)
               (if (> x 0) 1
                   (if (zero? x) 0
                       -1))))
```

L'utilisateur de if emboités n'est pas très lisible, et on peut vouloir une structure similaire aux switch des langages impératifs. On utilise alors la forme (cond ($condition_1$ $corps_1$)
($condition_2$ $corps_2$)
…
($condition_n$ $corps_n$)

```
Exemple :
(define signe
  (lambda (x)
    (cond ((> x 0) 1)
          ((zero? x) 0)      ; la fonction signe est tout à fait la même que la précédente avec les if imbriqués
          (else -1))))       ; si tous les cas fait, penser à mettre juste un else pour gagner en temps de calcul
```

On utilise and et or pour connecter différentes parties d'une expression de test.

```
Exemple :  (define fraction-egyptienne?      ; une fraction égyptienne est du type 1/n
             (lambda (x)
               (and (rational? x) (exact? x) (= (numerator x) 1))))      ; une lambda peut renvoyer un booléen
```

Exemples divers :

```
(define randomInt     ; soit 0<=a<=b.
  (lambda (a b)
    (+ (random (+ (- b a) 1)) a)))
; renvoit un nombre au hasard dans [a, b]

(define naturel?
  (lambda (x)
    (and (integer? x) (>= x 0))))
```

IV. Variables locales : let et let*

Pour déclarer des variables locales, on utilise principalement la forme (let ((var_1 $expression_1$) … (var_k $expression_k$)) corps)
On utilise ceci pour ne pas avoir à calculer plusieurs fois la même chose. On simplifie l'expression du calcul tout en gagnant en temps. Ainsi, pour le déterminant d'un polynôme du second degré, nous écrivons :

```
(define racine1    ; une racine de ax²+bx+c, a≠0
 (lambda (a b c)
  (let ((delta (discriminant a b c)))
    (/ (- (sqrt delta) b) (* 2 a)))))
```

```
(define discriminant
      (lambda (a b c)
            (- (sqr b) (* 4 a c))))
```

Attention : les variables déclarées avec un let sont toutes montées en même temps !

```
(define exemple
 (lambda ()
  (define x 10)          ; on déclare une variable x égale à 10. Utiliser define pour ça est sale…
  (let ((x (+ x 5)) (y x))     ; on déclare une variable locale x égale à x + 5 soit 15
   (+ (* 2 x) y))))            ; le (y x) fera bien y = x, mais en référence au x initialisé à 10 !
```

Si on veut que les variables soient évaluées dans l'ordre (de gauche à droite), on utilise let*.

```
(let ((x 5))              ; on déclare x à 5
 (let*  ((y (+ x 10))     ; soit y à x + 10, soit 15
         (z (* x y))      ; soit z à x * y. On utilise la valeur précédente de y, donc 5*15 = 75
    (+ x y z)))           ; 5 + 15 + 75 = 95
```

Un exemple d'intérêt des variables locales pour un calcul avec des termes qui reviennent souvent :

```
(define exemple          ; on calcule  √(x²+y²) - √(x²-y²)
 (lambda (x y)           ;             1+√(x²+y²) + √(x²-y²)
  (let* ( (x2 (* x x))
          (y2 (* y y))
          (add (sqrt(+ x2 y2)))        ; on utilise les valeurs précédentes de x2 et y2 : il faut un let*
          (sub (sqrt(- x2 y2))) )
  (/ (- add sub) (+ 1 add sub))) ))    ; attention à bien mettre le calcul final entre parenthèses !
```

Si nous sommes dans un Scheme qui ne possède ni let, ni let*, on peut toujours les créer à partir de lambda. Remarquons déjà que let* équivaut à plusieurs let imbriqués :

```
(let* ((x 4) (y (+ x 1)))        ⟺    (let ((x 4))
 (+ x y))                              (let ((y (+ x 1)))
                                        (+ x y)))
```

Attention ! Dans un let, chaque composante peut-être évaluée par un processeur différent, mais le let doit tout parcourir.*

De plus on remarque qu'on peut écrire le let avec un lambda :

```
(let ((x 2) (y 3)) (+ x y))      ⟺    ((lambda (x y) (+ x y)) 2 3)
```

modèle ((lambda (var1 var2 …) body) value1 value2 …)

Donc on arrive à faire le let* uniquement avec des lambda :

```
(let* ((x 4) (y (+ x 1)))        ⟺    ((lambda (x)
 (+ x y))                              ((lambda (y) (+ x y)) (+ x 1))) 4)
```

Enfin, une fonction importante :

```
(define las-vegas        ; cette fonction tire au hasard les nombres 3, 5, 7 avec des probabilités définies
 (lambda ()
  (let ((x (random 7)))          ; soit un nombre aléatoire...
   (cond ((<= x 3) 5)            ; on le contraint à 4/7 à être 5
         ((<= x 5) 7)            ; on le contraint à 2/7 à être 7
         (else 3)))))            ; on le contraint à 1/7 à être 3
```

V. Itératif et récursif

Vérifions que la probabilité de tirer un 2 avec le dé de las-vegas défini précédemment soit bien 2/7.
Autrement dit, on effectue 2000 lancers et on compte le nombre de 2 obtenus.
Il y a deux solutions pour cela : itérative ou récursive enveloppée (qui est la seule vrai récursivité !).
Regardons la solution récursive enveloppée :

```
(define (test-las-vegas)
 (let ((n 2000))                    ; le nombre d'expériences
  (define (nbFois2 nbLancers)       ; une sous-fonction ne peut-être que immédiatement sou un let ou
   (if (zero? nbLancers)            ; sous un define. Ce sont les deux seuls cas !
      0         ;s'il n'y a plus de lancers à faire, on s'arrête. On ne lance plus rien : on obtient 0 fois deux
      (let ((dé (las-vegas)))       ; on lance le dé
       (if (= dé 2)                 ; si on obtient un deux
          (+ 1 (nbFois2 (- nbLancers 1)))    ; alors le nombre de fois qu'on a obtenu un deux augmente
          (nbFois2 (- nbLancers 1))))))      ; sinon il est le même, et on relance le dé
  (exact->inexact (/ (nbFois2 n) n))))       ; a la fin on veut un nombre à virgule, pas une fraction
```

Le fait qu'il y ait ce + 1 devant (nbFois2 (- nbLancers 1))), nous montre que la récursivité n'est pas terminale. C'est donc un véritable cas de récursivité.
En revanche, le (nbFois2 (- nbLancers1))))) est une récursivité terminale car précédé de rien : le compilateur nous garantit qu'il sera interprété comme une boucle while.
Comme ici dans un cas on a de la récursif et dans l'autre cas un while, c'est globalement récursif.

Le modèle d'une solution itérative est le suivant :

```
(define (test-las-vegas)
 (define (iter "variables de boucle")   ; on défini une fonction itérative à l'intérieur avec ses variables
  (if "fini"                            ; on regarde si on a fini
     "le résultat"                      ; si oui, on renvoie le résultat
     (iter "amélioration des variables de boucle"))) ; si non, on fait évoluer les variables de boucle
 (iter "initialisation des variables de boucle"))    ; on initialise la boucle : c'est le starter
```

Par exemple, la version itérative du test de las-vegas est la suivante :

```
(define (test-las-vegas)
 (define (iter n acc)                   ; soit n le nombre d'expériences restantes et acc le nombre de 2 tirés
  (if (zero? n)                         ; si j'ai fini…
     (/ acc 2000.0)                     ; je renvoie le résultat avec un exact->inexact par virgule
     (let ((dé (las-vegas)))            ; sinon, soit le résultat de l'expérience…
      (if (= dé 2)                      ; …si on tire un 2, alors
         (iter (- n 1) (+ acc 1))       ; … le nombre de 2 augmente
         (iter (- n 1) acc)))))         ; et si on ne tire pas un 2, on en a toujours le même nombre
 (iter 2000 0))                         ; on a 2000 lancers à faire et au départ on a tiré aucun 2, starter !
```

On a (iter qui n'est précédé de rien, c'est une récursivité terminale et le compilateur nous assure que ce sera une boucle while. Ceci n'est pas vrai pour tous les langages. Certains ne détectent pas qu'il ne s'agit pas d'un mécanisme itératif, et mettent en place une pile.
Par exemple, dans :

```
public void foo(){
        System.out.println(« Hello ») ;
        Foo();
}
```

En C le même code fait crasher la machine sauf si on enclenche l'optimisation de niveau 9 avec –O9, qui va chercher à détecter récursif/itératif.

Java ne voit pas que ceci est itératif et met en place une pile qui va exploser..

```
(define (foo)
        (printf « Hello\n »)
                ( foo ))        ; Scheme détecte que c'est itératif, ne met pas de pile et tourne bien
```

Regardons par exemple comment écrire la multiplication, de façon itérative ou récursive.

$X * Y = X + X*(Y-1)$ *; 5 * 4 = 5 + 5*3 = 5 + 5 + 5*2 = 5 + 5 + 5 + 5*1 = 5 + 5 + 5 + 5 + 0 = 25*
$X * 0 = 0$

Et on donne l'algorithme récursif écrit en scheme :

```
(define (mul x y)
  (if (zero? y)
      0
      (+ (mul x (- y 1)) x)))      ; ceci est récursif enveloppé à cause du +, donc de la véritable récursivité !
```

A présent, la version itérative sur le modèle défini plus haut :

```
(define (mulit x y)
  (define (iter result b)          ; la partie itérative de la chose avec un compteur 'résult' qui additionne
    (if (zero? b)                  ; si on a fini, alors…
        result                     ; …on renvoie le résultat
        (iter (+ result x) (- b 1)) )) ; on rajoute x dans le compteur et on décrémente b
  (iter 0 y)) ; starter : au départ le compteur est nul, et on transmet y. C'est un choix de transmettre y !
```

Cette dernière forme est de la récursivité terminale, autrement dit : de l'itératif. Notons bien qu'il n'y a pas besoin de parenthèses pour passer les arguments !

Exemple d'itératif : le produit de tous les entiers de l'intervalle [a, b].

```
(define (interfac a b)             ; On fait a*(a+1)*(a+2)*…*b
  (define (iter acc boucle)        ;on le fait de façon itérative avec un accumulateur et une variable de boucle
    (if (> boucle b)               ; on fait monter a. S'il dépasse b, on s'arrête.
        acc                        ; on renvoie l'accumulateur
        (iter (* acc boucle) (+ boucle 1)) ))    ; sinon, on multiplie l'accumulateur par a et on incrémente a
  (iter 1 a))                      ; pour multiplier, l'accumulateur doit être à 1 au départ (le neutre)
```

Un autre exemple : calculer l'intégrale d'une fonction.

```
(define (intégrale f a b dx) ; on passe la fonction f en paramètre, l'intervalle [a, b] et le pas d'intégration
  (define (iter result progress)  ; 'progress' contient a auquel on ajoute à chaque fois le pas d'intégration
    (if (> progress b)            ; si on a fini l'intervalle, on renvoie le résultat
        result
        (iter (+ result (* dx (f progress))) (+ progress dx)))) ; sinon ajoute dx.f(point courant) au résultat
  (iter 0 a))      ; au départ le résultat de l'intégration est 0.
```

Regardons comment faire une puissance a^b par dichotomie, i.e. en coupant le problème en deux.
On a $2^{10} = (2^2)^5$ et $2^{11} = 2.(2^2)^5$
Donc si b est pair, on fait $(a^2)^{b/2}$ et si b est impair on fait $a.(a^2)^{b/2}$
D'où une écriture récursive :

```
(define ($expt a b)
  (cond ((zero? b) 1)
        ((even? b) ($expt (sqr a) (quotient b 2)))          ; dans les entiers, on utilise le quotient !
        (else (* a ($expt (sqr a) (quotient b 2))))))
```

On doit faire rentrer le '* a' si on veut une écriture itérative. Ainsi :

```
(define ($expt a b)
  (define (iter z p q)                                      ; calcule z.p^q
    (cond ((zero? q) z)
          ((even? q) (iter z (sqr p) (quotient q 2)))
          (else (iter (* z p) (sqr p) (quotient q 2)))))    ; la différence : on multiplie z par p
  (iter 1 a b))
```

VI. Obtenir la trace d'une fonction

On peut espionner une fonction avec des instructions d'affichages en manuel, ce qui est un peu sale…

```
(define (mul x y)
  (printf "Appel avec ~a * ~a\n" x y)
  (if (zero? y)
      0
      (+ (mul x (- y 1)) x)))
```

```
> (mul 5 5)
Appel avec 5 * 5
Appel avec 5 * 4
Appel avec 5 * 3
Appel avec 5 * 2
Appel avec 5 * 1
Appel avec 5 * 0
25
```

On préfère utiliser la bibliothèque de trace : on l'importe, on met la fonction en écoute et on exécute.

```
(define (mul x y)
  (if (zero? y)
      0
      (+ (mul x (- y 1)) x)))

(require (lib "trace.ss"))
(trace mul)
(mul 5 5)
```

```
(mul)
|(mul 5 5)
| (mul 5 4)
| |(mul 5 3)
| | (mul 5 2)
| | |(mul 5 1)
| | | (mul 5 0)
| | | 0
| | |5
| | 10
| |15
| 20
|25
25
```

On voit clairement avec la remontée dans la pile que cette fonction est récursive.
Pour tracer une fonction itérative, il faut sortir les fonctions internes à espionner. Ainsi, on décompose la fonction avec son entrée et le starter d'un côté, la partie itérative de l'autre. D'où :

```
(define (iter2 x result b) ; partie itérative
   (if (zero? b)
      result
      (iter2 x (+ result x) (- b 1)) ))

(define (launcher x b)   ; entrée et starter
  (iter2 x 0 b))

(display "Trace de la version itérative")
(trace iter2)
(launcher 5 5)
```

```
(iter2)
|(iter2 5 0 5)
| (iter2 5 5 4)
| |(iter2 5 10 3)
| | (iter2 5 15 2)
| | |(iter2 5 20 1)
| | | (iter2 5 25 0)
| | | 25
| | |25
| | 25
| |25
| 25
|25
25
```

On voit à la remontée qu'il ne s'agit pas d'une fonction récursive car les résultats sont tous identiques : c'est caractéristique d'une itération !

VII. Un aperçu du style CPS

On peut définir un calcul par une opération et sa suite. Par exemple, dans la factorielle :

```
(define (fac n)          ; on défini n!
  (if (zero? n)          ; quand on a 0, on s'arrête avec 0! = 1
      1
      (* n (fac (- n 1))))) ; sinon on fait r*n. L'appel récursif est continué par la fonction r |-> r*n
```

On va généraliser le calcul en faisant f(n!) où f est la continuation du calcul. Avec f = id, on retrouvera n! en cas particulier. La fonction sera dite sous forme CPS (Continuation Passing Style) :

```
(define (k-fac n f)      ; dans une fonction CPS, on se définit en paramètre supplémentaire la continuité
  (if (zero? n)          ; si n vaut 0, c'est toujours le cas d'arrêt. On applique la fonction sur 1
      (f 1)
      (k-fac (- n 1) (lambda (r) (f (* r n))))) ; sinon, c'est r |-> f(r*n), comme on l'a vu précédemment
```

Ainsi, (k-fac 5 id) donne bien 120 tandis que (k-fac 5 (lambda (x) (* x 3))) donne 360…

VIII. Listes : constructeur, accesseur, reconnaisseur

La liste, comme toute structurée de données, dispose de trois familles de fonctions : constructeur, accesseur, reconnaisseur.
Pour construire une liste, on utilise la fonction list ; quelques exemples :

```
(list 10 12 14 16)          ; répond (10 12 14 16)
(list (> 10 12) (not #f))   ; répond #f #t. Les expressions SONT évalués ; exemple (+ 1 2) donnera 3
(list)                      ; répond ()[2] pour la liste vide
```

On peut vouloir mettre des données dedans, comme des lettres. Or, si on passe une lettre tel quel, elle risque d'être prise pour une variable et son évaluation conduira à une erreur.

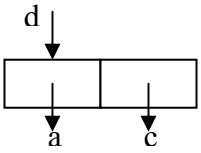
```
(list a b)                  ; répond que la variable a est indéfinie
```

Pour cela, on demande de ne pas évaluer certaines choses, grâce à quote (citation) :

```
(list (quote a) (quote b))  ; répond (a b)
```

Comme le quote est très utile, on l'abrège avec l'accent aigu[3] : (list 'a 'b).

Un constructeur plus primitif (à partir duquel on peut définir la fonction list) est cons. Alors que list est n-aire, cons est strictement binaire et construit la liste par paires (doublets).
On peut lui donner un terme et une liste, par exemple (cons 10 (list 20 30)) *; répond (10 20 30)*
On peut aussi monter intégralement la liste avec cons. Regardons alors le fonctionnement en doublets.



Un doublet est un pointeur vers une zone comportant deux mots mémoire ; il occupe donc toujours 2*4 = 8 octets en mémoire.
L'exemple ci-contre montre le résultat de (define d (cons 'a 'b))
(define big (cons (cons 1 2) (cons 3 (cons 4 5))))

Lorsqu'on affiche la dernière construction, on s'attendrait à ((1 . 2) . (3 . (4 . 5))).
Or ce n'est pas le résultat ; en effet, on récupère ((1 . 2) 3 4 . 5) ; pourquoi ?
On utilise la convention point-parenthèse : chaque fois qu'il y a une point suivi d'une parenthèse ouvrante, on les supprime ainsi que la parenthèse fermante qui s'y rapporte. Ainsi, en étapes :
((1 . 2) . (3 . (4 . 5))) → ((1 . 2) 3 . (4 . 5)) → ((1 . 2) 3 4 . 5)

Pour accéder aux composantes, on a deux fonctions :

- car [4], qui rend le premier élément.
- cdr. Dans une liste on dit qu'il rend la liste privée de son premier élément, et dans un doublet c'est la 2nd composante.

Notons bien qu'on ne peut pas rendre le premier ou second élément d'un ensemble vide (tester null? L).
Quelques exemples :

```
(define maListe (list 10 20 30))          ; (10 20 30)
(car maListe)                             ; 10
(cdr maListe)                             ; (20 30)
(define autreListe (cons 10 (cons 20 30))) ; (10 20 . 30)
(car autreListe)                          ; 10
(cdr autreListe)                          ; (20 . 30)
```

On peut considérer car et cdr comme des inverses de cons, dans le sens où ce que cons construit, car et cdr peuvent le défaire ; de même, ce que car et cdr défont, cons peut le construire.

Enfin, le reconnaisseur d'une liste est pair? tel que « pair? obj = #t ↔ obj a un car ^ obj a un cdr ».
Attention, au sens de cette définition, l'ensemble vide n'ayant ni car ni cdr, (pair? (list)) donne #f !

Autre remarque : null? nous dira que la liste est vide si elle ne contient aucun terme ; dès qu'il y en a un, elle est non vide. Donc, on ne peut pas se servir de null? pour avoir l'existence d'un cdr !

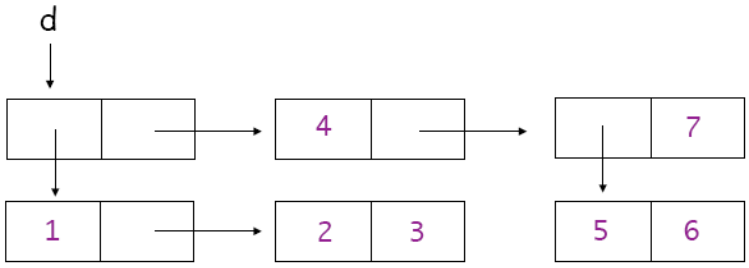
IX. Listes : représentation

Nous avons vu précédemment la convention du point-parenthèse. Se rajoute à cela le fait qu'une boîte dans un car sort à la verticale et à une boîte dans un cdr à l'horizontale. On peut maintenant donner l'algorithme qui permet de représenter un chaînage de doublets par son affichage[5] :

- une flèche verticale + une boîte → une parenthèse ouvrante
- un élément dans un car → on affiche 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

Exemple :

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



Trace de l'algorithme :

- une flèche verticale + une boîte : (
- un car, je vais dans le car. Flèche verticale plus boîte : ((1
- un cdr, je vais dans le cdr. Flèche horizontale plus boîte : ((1 2
- un cdr, je vais dans le cdr. J'affiche le point, je clôture : ((1 2 . 3)
- On remonte. On va dans le cdr. Flèche horizontale plus boîte : ((1 2 . 3) 4
- Un car, je vais dans le car. Flèche horizontale plus boîte : ((1 2 . 3) 4 (5
- Un cdr, je vais dans le cdr. J'affiche le point, je clôture : ((1 2 . 3) 4 (5 . 6)
- On remonte. Un cdr, je vais dans le cdr. J'affiche le point, je clôture : ((1 2 . 3) 4 (5 . 6) . 7)

Lorsque le cheminement qu'on fait dans la liste est plus complexe, on utilise des notations contractées. Par exemple, si on veut récupérer 2 de la figure précédente : (car(cdr(car d))) s'abrège en (cadar d). On met un c au début, un r à la fin, et entre les deux d si on veut un cdr, et a si on veut un car. Attention, au final, cela s'écrit à l'envers !

Exemples sur la figure précédente :

```
(cadar d)      ; on prend le car de d : (1 2 . 3). On prend le cdr : (2 . 3). On prend le car : 2
(caaddr d)     ; on prend le cdr de d : (4 (5 . 6) . 7). On prend le cdr : ((5 . 6) . 7). On prend le car : (5 . 6)
```

Et inversement, si on veut 3 :

- c'est le cdr de sa boîte.
- sa boîte est le cdr de la précédente.
- laquelle est le car de d.

D'où (car(cdr (cdr d))) soit (cddar d) en recopiant à l'envers.

X. Récurrence sur les listes

On veut la somme d'une liste de nombres.

```
(define (somme L) ; de même que x est un réel et n un entier, L désigne la liste en Scheme
  (if (null? L)
      0 ; la somme d'une liste qui ne contient rien c'est 0 : le neutre de l'addition
      (+ (car L) (somme (cdr L))))) ; sinon, c'est le premier terme plus la somme des termes restants
```

Et demander les termes « restants », cela revient à demander « tout L sauf son premier terme » : le cdr. On peut l'écrire de façon symétrique :

```
(define (autreSomme L)
  (if (pair? L)
      (+ (car L) (autreSomme (cdr L))) ; la somme d'un doublet, c'est le premier élément plus les autres
      0)) ; si ce n'est pas un doublet, cela veut dire que le car et le cdr n'existent plus : la liste vide
```

Notons que (pair? (list 1)) répond #t : le cdr est vide, mais il a quand même une existence. Donc pas vide.

De même, regardons la taille d'une liste :

```
(define (longueur L)
  (if (pair? L)
      (+ 1 (longueur (cdr L))) ; récursif
      0))

(define (i-longueur n L)
  (if (pair? L)
      (i-longueur (+ n 1) (cdr L)) ; itératif
      n))

(define (starter L) ; un starter pour l'itératif
  (i-longueur 0 L))
```

```
(longueur)
|(longueur (a b c))
| (longueur (b c))
| |(longueur (c))
| | (longueur ())
| | 0
| |1
| 2
|3
3
```

```
(k-longueur)
|(k-longueur 0 (a b c))
| (k-longueur 1 (b c))
| |(k-longueur 2 (c))
| | (k-longueur 3 ())
| | 3
| |3
| 3
|3
3
```

La taille d'une liste L est 1 + la taille de la liste L privée de son premier élément ; enfin, la taille d'une liste vide est 0. Ceci se transcrit tout seul. Si nous avons une liste non-vide (pair?) alors on rajoute 1 et on recommence en enlevant le premier élément (cdr) ; sinon, on n'ajoute rien.

Enfin, un dernier exemple pour montrer comment rendre une liste comme résultat :

```
(define (duplicata L) ; créée un double de la liste d'entrée
  (if (pair? L)
      (cons (car L) (duplicata (cdr L))) ; avec cons on reconstruit une liste ayant le 1er élément et la suite
      (list))) ; si on a une liste vide, on rend une liste vide
```

Résumons après tous ces exemples le modèle de récurrence simple sur les listes :

```
(define (fRec L) ; soit une fonction récursive sur les listes
  (if (pair? L)    ; si notre liste n'est pas la liste vide, alors…
      (combinaison (car L) (fRec (cdr L))) ; on combine le premier élément de la liste avec les suivants
      (cas-liste-vide))) ; et si la liste est la liste vide, on rend le cas adapté, i.e. le résultat du cas d'arrêt
```

On peut maintenant passer à des cas plus subtils. Par exemple : comment ajouter un élément en fin ?

```
(define (ajout-en-fin L x)
  (if (pair? L)
      (cons (car L) (ajout-en-fin (cdr L) x))      ; on recopie la liste
      (list x)))                                   ; et une fois tout recopié, on met l'élément à rajouter en fin
```

Et maintenant, on en arrive à définir la fonction append qui concatène deux listes :

```
(define ($append L1 L2)        ; on utilise le $ pour différencier de la primitive append
  (if (pair? L1)
      (cons (car L1) ($append (cdr L1) L2)) ; le résultat est la liste débutant avec le 1er élément de L1
      L2)) ; … et dont le reste est la concaténation du reste de L1 avec L2. C'est donc en Θ( |L1| )
```

Un exemple encore un peu plus complexe : attribuer à chaque élément de la liste la somme des suivants.

```
(define (somme-cumulee L) ; (list 1 2 3) -> (list (+ 1 2 3) (+ 2 3) 3)
  (if (pair? L)    ; est-ce que la liste est vide ?
      (if (pair? (cdr L))    ; elle n'est pas vide : a-t-elle un seul élément ou 'plusieurs' ?
          (let ((reste-fait (somme-cumulee (cdr L))))        ; soit la somme cumulée des éléments suivants
            (cons (+ (car L) (car reste-fait)) reste-fait))       ; on la rajoute au premier élément de L
          L)                ; si la liste n'a qu'un seul élément, c'est lui le résultat
      (list)))       ; si c'est la liste vide, alors on renvoit vide.
```

Enfin, quelques différences entre les constructeurs pour éviter les pièges classiques :

- (cons '(a b c) 'd) va construire un bloc avec (a b c) et un autre avec d. Pas de problème.
- (append '(a b c) 'd) est une erreur car append prend uniquement deux listes, d n'en est pas une.
- (list '(a b c) 'd) fabrique une liste en extension. [6]

XI. Corrections d'exercices en Scheme

1) Inverser une liste

```
(define ($reverse L)   ; on met un $ pour différencier de la primitive
  (if (null? L)        ; a-t-on fini d'inverser la liste ?
      null             ; si oui, on la renvoie. C'est le traitement de la liste vide.
      (append ($reverse (cdr L)) (list(car L))))) ; sinon, on met le premier élément à la fin et on continue
```

Le petit détail ici est que append doit prendre deux listes comme argument, et car L n'en est pas nécessairement une. On l'englobe donc d'un list.
Le principe est que pour inverser une liste $L_1\ L_2\ L_3\ \dots\ L_n$ c'est :

- si la liste est vide, c'est également la liste vide
- sinon, c'est \$reverse($L_2\ L_3\ \dots\ L_n$) **.** L_1

Autrement dit en notation Kounalis : $r(\emptyset) = \emptyset$; $r(x.L) = r(L).x$, où **.** signifie la concaténation

Pour évaluer la complexité de \$reverse, il faut avant tout savoir ce que l'on mesure. Ici, l'opération la plus coûteuse sera les appels à cons, au travers de append. Ainsi $C_0 = 0$ puisque cons n'a pas été appelé, et :
$C_n = C_{n-1} + 1 + (n-1)$, car : - (\$reverse (cdr L). Il faut inverser la liste à un élément de moins, d'où C_{n-1}.
- append : il faut greffer la liste inversée avec l'élément courant, d'où n-1
- list(car L). Cela utilise un appel à cons. (list α) coûte un cons, (list α β) 2, etc.

On a $C_n = C_{n-1} + 1 + (n-1)$ qu'on simplifie en $C_n = C_{n-1} + n$, pour avoir la même complexité asymptotique.
Il ne reste plus qu'à résoudre l'équation en quelques commandes Maple[7] :

```
rsolve({c(0)=0,c(n)=c(n-1)+n},c(n))
→ (n+1)(n/2 + 1) – 1 –n
simplify(‘’) ;
→ n²/2 + n/2
factor(‘’) ;
→ (n(n+1))/2
```

D'où une fonction reverse en $\Theta(n^2)$.
On complète ceci par une fonction permettant comme toujours de faire un test de validité :

```
(if (not (equal? ($reverse '(a b c d)) '(d c b a)))
   (error "Votre fonction $reverse est fausse !"))
```

A présent, si on voulait faire une version itérative de cette fonction ? En arrivant à enlever le append, on se retrouverait avec un algorithme linéaire, d'où un certain intérêt. D'où :

```
(define ($reverse-iter L)
  (define (iter L acc)                    ; on se donne un accumulateur pour stocker la liste inversée
    (if (null? L)                         ; si on a fini d'inverser la liste, alors…
        acc                               ; on renvoie la liste inversée.
        (iter (cdr L) (cons (car L) acc)))) ; sinon… ligne principale de traitement
  (iter L ()))
```

La ligne principale permet de continuer le traitement sur la suite de L (avec cdr L), et d'ajouter à la liste d'accumulation l'élément que l'on enlève. Ainsi, le nombre d'appels à cons est très visiblement linéaire.

2) Exponentiation modulaire

```
(define (exptmod a b n) ; a^b modulo n
  (cond ((zero? b) 1)       ; si je fais a^0 alors c'est 1
        ((even? b) (exptmod (modulo (sqr a) n) (quotient b 2) n)) ; si b est pair, alors c'est (a²)^b/2
        (else (modulo (* a (exptmod (modulo (sqr a) n) (quotient b 2)  n)) n)))) ; sinon c'est a. (a²)^b/2
```

Cet algorithme fait une dichotomie sur la puissance et un modulo à chaque étape pour réduire les calculs.

3) Test de primalité de Fermat

Rappel : on dit que n passe le test de Fermat en base a si $a^{n-1} \equiv 1$ [n]. Le test de Fermat pour savoir si n est [presque certainement] premier consiste à faire plusieurs fois l'expérience suivante : on tire au hasard une base a dans [2,n-1] première avec n, et on regarde si n passe le test en base a. S'il ne le passe pas, il est composé, sinon il y a de grandes chances qu'il soit premier. Nous nous contenterons de trois tests, avec les bases a=2, a=3 et une base aléatoire dans [2,n-1] première avec n.

On commence par écrire la fonction qui retourne un nombre aléatoire de [2, n-1] premier avec n :

```
(define random-base     ; il est dit que 0<=a<=b. Philosophie du scheme : l'utilisateur se débrouille avec.
 (lambda ( n )
  (let ((test (+ (random (- n 2)) 2)))     ; soit un nombre aléatoire de [2, n-1]
   (if (equal? (GCD test n) 1)             ; s'il est premier avec n alors c'est mon résultat
      test
      (random-base n)))))                  ; sinon je cherche à en fabriquer un autre
```

A titre de curiosité, la probabilité que 2 entiers au hasard soient premiers est de $6 / \pi^2$ d'après les séries.
A présent, il n'y a plus qu'à faire le test de fermat :

```
(define (fermat-premier? n)     ; c'est très léger de tester sur 3 bases mais % de chance est déjà très élevé
 (define (ok-en-base? a)                ; fonction interne pour vérifier si a^(n-1) ≡ 1 [n]
  (= (exptmod a (- n 1) n ) 1))
 (and (>= n 2)                          ; Ca doit être supérieur ou égal à 2 pour tester.
      (or (= n 2)                       ; le test de Fermat échoue pour 2 : on patch !
          (= n 3)                       ; pareil
          (and (ok-en-base? 2)                      ; si le test passe en base 2…
               (ok-en-base? 3)                      ; …ainsi qu'en base 3…
               (ok-en-base? (random-base n))))))    ; … et dans une base au hasard, alors c'est quasi sûr
```

Maintenant, sachant un nombre, on peut rechercher le plus petit nombre premier supérieur ou égal :

```
(define (ppp>= n)               ; plus petit premier supérieur ou égal à n
 (define (iter n)               ; on fait progresser ce n jusqu'à ce que fermat-premier confirme
  (if (fermat-premier? n)       ;on sait qu'on finira bien par arriver un jour sur un nombre premier à ce test
     n
     (iter (+ n 2))))           ; les nombres pairs ne sont pas premiers, on les saute
 (iter (if (odd? n) n (+ n 1))))  ; donc si n est impair (odd?) on commence à n et sinon à n+1
```

Par exemple, pour savoir à quelle distance de 10^{50} le plus petit premier se situe :

```
(let* ((start (expt 10 50)) (sol (ppp>= start))) ; le plus petit premier supérieur à 10^50 est 10^50 + 151.
  (printf "le plus petit premier supérieur à 10^50 est 10^50 + ~a\n"
       (- sol start)))
```

Enfin, on va regarder là où le test de fermat se trompe par rapport à une fonction classique de test.

```
(define (premier? n)
 (define (ppdiv n)
  (define (iter k)
   (cond ((> (sqr k) n) n)
         ((zero? (modulo n k)) k)
         (else (iter (+ k 2)))))
  (if (even? n) 2 (iter 3)))
 (and (>= n 2) (= n (ppdiv n))))
```

```
(define (carmichael n)
 (define (iter n)
  (if (> n 10000)
     (printf "\n")
     (let ((b1 (premier? n)) (b2 (fermat-premier? n)))
      (if (not (equal? b1 b2))
         (printf "n = ~a, premier? = ~a, fermat-premier = ~a\n" n b1 b2)
         (iter (+ n 2))))))
 (iter n))
```

4) Formes spéciales

Une personne prétend pouvoir coder le 'if' à l'aider de cond, de la façon suivante :

```
(define (new-if predicate then-clause else-clause)
 (cond (predicate then-clause)
       (else else-clause)))
```

Le programme est démontré sur deux exemples :

```
(new-if (= 2 3) 0 5)    ; rend bien 5
(new-if (= 1 1) 0 5)    ; rend bien 0
```

A présent, on veut utiliser cette nouvelle version du if sur le cas de la factorielle ci-dessous :

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

Que va-t-il se passer quand on fera (fac 5) ?
Toute fonction travaille en ordre applicatif : on évalue la fonction et ses arguments (dans un ordre quelconque), puis on applique la valeur de la fonction aux valeurs des arguments.
Ici, la clause 'sinon' contituée de « (* n (fac (- n 1))) » est un des arguments du new-if : on cherchera donc à l'évaluer. Et on va ainsi boucler indéfiniment…

Il existe donc des formes « spéciales » qui n'évaluent pas nécessairement tous leurs arguments et n'ont pas de règle uniforme. C'est le cas de : define, lambda, if, cond, and, quote…
Exemple : (define hello 'bonjour). On ne peut pas chercher à évaluer hello : il n'a pas encore de valeur !
[i]

---

[1] Il y a une différence entre fonctions et procédures. Examinons f(x) = 3x + 6 et g(x) = 3(x+2).
Une fonction est un objet mathématiquement qui prend des données d'entrées et rend un résultat. f(x) et g(x) prennent les mêmes entrées et donnent le même résultat : ce sont donc les mêmes fonctions.
Une procédure est une suite d'instruction qui définit la façon de calculer la chose. Pour f(x), on multiplie x par 3 puis on y ajoute 6 ; pour g(x) on ajoute 2 à x puis on multiplie le tout par 3. Ce sont donc des procédures différentes.
f(x) et g(x) sont deux procédures différentes qui implémentent une même fonction. *[Brian Harvey, cours de Berkeley]*

[2] La norme Scheme impose de noter l'ensemble vide par '(). Mais DrScheme, qui a sa façon de faire, accepte le 'null' à la façon de Java, en mettant directement ().

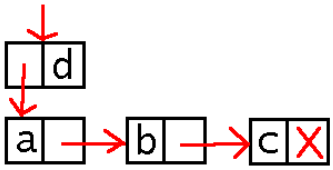
[3] Le quote fait un mot à l'état brut, ce n'est pas une string. Ca signale juste « n'évalue pas ça, rend le moi tel quel ».

[4] Les mots CAR et CDR viennent de la première implémentation de Lisp par Steve Russel en 1959, sur l'IBM-704 qui avait entre autres deux registres : Adress et Decrement. D'où CAR = Content of Adress Register, et CDR = Content of Decrement Register. *[John McCarthy, histoire du Lisp, Stanford]*

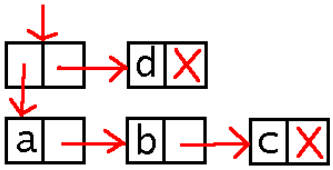
[5] On peut aussi voir qu'une liste n'est rien d'autre qu'un arbre binaire où « car » est le fils gauche et « cdr » le fils droit, avec 'cons' en racine. Avec l'algorithme proposé par Nils M Holm sur comp.lang.scheme (2006), on peut faire afficher les boîtes avec leur contenu d'après une expression sur la convention du point-parenthèse ; la fonction est nommée draw-tree.

[6] Regardons la différence sur un exemple :

cons '(a b c) 'd



list '(a b c) 'd



[7] Pour résoudre 'à vue' cette équation de récurrence, on peut considérer que c'est assez similaire à une expression qu'on veut primitiver. Par exemple $C_n = C_{n-1} + n^2$ est, à vue, en n^3 car la primitive de n^2 est de l'ordre de n^3. Donc dans notre cas de $C_n = C_{n-1} + n$, on peut conclure à un $O(n^2)$ sans réfléchir excessivement.