Le langage Scheme : Avancé

D'après les cours de J.-P. Roy

I. Primitives indispensables

Pour un style impératif	*Résultats*
(begin e_1 … e_n)	Exécute de gauche à droite les instructions e_1 … e_n, et rend le résultat de e_n
(set! symbole expression)	Une forme spéciale. La variable associée au symbole devient la valeur de l'expression. Par exemple, (set! v 3) met à 3 la variable v.
(set-car! s expr)	Pareil que set! et on fait muter le car d'une liste.
(set-cdr! s expr)	Pareil que set! et on fait muter le cdr d'une liste.
(do ((x_1 i_1 s_1) … (x_n i_n s_n) (test? e_1 ... e_n) c_1 … c_n)	On pose les variables x_i initialisées à i_i et on décrit leur évolution s_i. Si le test est vérifié, alors on fait les instructions e_i associées et on s'arrête. Sinon à chaque tour on fait les instructions c_i. Un do 'impératif' est un do qui contient des instructions c_i (corps de boucle)
(repeat n e_1 … e_k)	(dans turtle.scm) Repete n fois les instructions, et ne rend pas de résultat.

Primitive dans les tableaux	*Résultats*
(make-vector x)	Créer un tableau pouvant contenir éléments de type quelconque (nombres, pointeurs…). Initialisation à 0 par défaut. (define v (make-vector 10)) → #10(0). 10 éléments de valeur 0.
(vector L)	Créer un tableau en lui donnant son contenu. (define w (vector 'a (+ 1 2) '(a b c) 8 8 8). → #6(a 3 (a b c) 8) Il n'affiche le 8 qu'une fois car il détecte qu'il est répété. Plus compact.
(vector-ref Vecteur n)	Obtenir l'élément de numéro n du vecteur. Numérotation à partir de 0. Principe des tableaux : toute composante accessible en O(1).
(vector-set! Vecteur n)	Modifier la $n^{ième}$ composante d'un vecteur (elles sont toutes mutables !)
(vector-fill! Vecteur x)	Mettre x dans chaque composante du vecteur.
(vector-length Vecteur)	Obtenir la taille du vecteur.
(vector->list v) (list->vector v)	Passer d'un vecteur à une liste et réciproquement.

Entrée / sorties, ports…	*Remarques*
(void)	Toute fonction doit rentre quelque chose, c'est un 'rien' non affichable
(port? obj)	L'objet est-il un port ?
(current-output-port) (current-input-port)	Le port de sortie/entrée courant. On rappelle que les ports sont en Scheme des objets de $1^{ère}$ classe : passables en paramètres de fonction.
(output-port? obj) (input-port? obj)	L'objet est-il un port de sortie ? d'entrée ?
(read)	Lire un objet Scheme avec le port d'entrée. Ce qu'on tape est interprété.
(read-line)	Lire une suite de caractères et assembler en une chaîne sans fin de ligne.
(read-from-string str) (read-from-string-all str)	Lire des objets Scheme à partir d'une chaîne. Soit on prend le 1er objet Scheme complet et on s'arrête, soit on les prend tous et on rend une liste
(write x)	Afficher la valeur de x sous une forme lisible par (read)
(display x)	Afficher la valeur de x sous une forme lisible par… un humain
(printf <format-string> v …)	Sans résultat, non normalisé. Fonctionnement sur la même idée qu'en C. Format-string est composé de ~a (displAy) et ~s (write), et sauts \n
(format <format-string> v …)	Comme printf, mais on retourne la chaîne au lieu de l'afficher.
(open-[out/in]put-port f)	Ouvrir manuellement le port associé au fichier de chemin f.
(close-[out/in]put-port p)	Fermer le port (souvent obtenu par la primitive open-[out/in]put-port).
(call-with-[out/in]put-file f p [m])	Ouvrir (et fermer automatiquement ensuite) le fichier.
(eof-object? x)	Est-ce qu'on est arrivé à la fin du fichier ?

MCours.com

II. Fonctions d'arité variable

On rappelle que la primitive (apply f L) retourner le résultat de l'application de la fonction f à la liste L. Tout le fonctionnement de l'interprète Scheme est axé sur le couple eval/apply[1]. Son utilisation permet certains raccourcis ; par exemple, calculer le maximum d'une liste : (apply max '(2 3 4 6 5 7 9 8 1)). Utilisée avec d'autres fonctions d'ordre supérieur, elle calcule par exemple des séries en une ligne[2].

```
(define (fac n) (apply * (iota n 1)))        ; la factorielle est le produit des entiers de 1 à n

(define (e n) ; calcul de e^1 à l'ordre n, avec son développement en série entière : somme des 1/k!, de 0 à n
   (apply + (map (lambda (k) (/ 1.9 (fac k))) (iota n 0))))

; On peut même calculer en une ligne une approximation de π par la formule de Gregory
; (issue du développement de atan x), où π/4 = 1  - 1/3  + 1/5  - 1/7  + 1/9  - 1/11 + …
(define (pi n)
  (* (apply + (map (lambda (x) (* (expt -1 (+ x 1)) (/ 1.0 (- (* 2 x) 1)))) (iota (+ n 1) 1))) 4))

(define (integrale f a b n)
 (let ((d (/ (- b a) n)))
   (* (apply + (map f (iota n a d))) d)))

; calcul de vecteur : addition, loi externe, produit scalaire
(define (vect+ V1 V2)
  (map + V1 V2))

(define (ext* k V)
  (map (lambda (x) (* k x)) V))

(define (prodscal V1 V2)
  (apply + (map * V1 V2)))

; append recolle 2 listes. On veut enlever les
; parenthèses pour chaque élément de la A-liste.
; c'est comme si on voulait faire un append entre
; chaque association et le null !
(define (Aliste->liste AL)
 (apply append AL))

(define (liste->Aliste L) ; L a un nbr pair d'elements
 (if (null? L)
     L
     (cons (list (car L) (cadr L)) (liste->Aliste (cddr L)))))
```

La philosophie de Scheme est que si le langage se permet quelque chose, alors il doit l'offrir à l'utilisateur. On a des fonctions d'arités variable (ou n-aire) comme + et list, donc on peut en définir. On rappelle que le texte d'une fonction est constitué de doublets. On va donc donner une liste d'arguments, où l'on indique la présence de ceux qui nous intéressent, et on met . L pour signifier que L est une liste d'arguments optionnels.

```
(define (somcarres . L)                 ; fonction pour faire la somme du carré de ses arguments
   (if (null? L)
        0
        (+ (sqr (car L)) (apply somcarres (cdr L))))) ; et surtout pas (somcarres (cdr L)) qui est une liste !

(define (somcarres x y . L)                  ; maintenant, les deux premiers arguments sont obligatoires
   (define (recur L)
      (if (null? L)
           0
           (+ (sqr (car L)) (recur (cdr L)))))
   (+ (sqr x) (sqr y) (recur L)))
```

Programmons maintenant le minimax, intéressant en théorie des jeux sur l'élaboration des stratégies optimales. Il s'agit du plus petit maxima des listes passées en argument :

```
(define (minimax . L)
 (apply min (map (lambda (x) (apply max x)) L)))  ; je prend la liste des maxima, et j'en tire le minimum
```

Construisons la fonction iota acceptant 1, 2 ou 3 arguments. On rappelle que (iota n x s) signifie : une liste de n nombres commençant à x et espacés de s. Par défaut x = 0 et s = 1.

```
(define (MyIota n . L) ; idée : programmer une fonction la plus générale possible à l'intérieur et compléter
  (define (yoda n x s)
    (if (= n 0)
        '()
        (cons x (yoda (- n 1) (+ x s) s))))
  (cond ((null? L) (yoda n 0 1)) ; idée de thomas : faire les choses en partant de la fin, meilleure complexité
        ((null? (cdr L)) (yoda n (car L) 1))
        ((null? (cddr L)) (yoda n (car L) (cadr L)))
        (else (error "map est une fonction unaire, binaire, ternaire... "))))
```

Il est déconseillé d'utiliser length pour connaître la quantité d'argument car il peut coûter très cher en cas d'erreur de l'utilisateur. Il vaut mieux passer tester le car et le cdr à null, etc. etc., tout à la main.

```
(define ($list . L)   ; Un exercice un peu plus technique : programmer la primitive (list x ...) qui prend
  (append L '()))     ; un nombre arbitraire d'arguments (nul s'il le faut) et en fait une liste.
```

On utilise append qui procède au clonage des cellules de son premier argument, et accroche ceci aux cellules de la deuxième. Ainsi, on obtient bien un nouvel élément par le clonage.
On insiste : c'est très différent d'un append '() L, car on ne clone les cellules que du 1er argument !
Bien entendu, (define ($list . L) L) pourrait marcher mais on est conscient de ne cloner que le pointeur...

Enfin, on présente à nouveau le tri-fusion avec la fonction 'decouper' qui coupe la liste en deux de telle façon qu'on puisse immédiatement en extraite les 2 éléments pour continuer le traitement. Le mot continuer nous indique qu'il est indiqué de passer en CPS…

```
(define (k-decouper L f) ; une continuation f à 2 variables
  (cond ((null? L) (f L L))
        ((null? (cdr L)) (k-decouper (cdr L) (lambda (L1 L2) (f (cons (car L) L1) L2))))
        (else (k-decouper (cddr L) (lambda (L1 L2) (f (cons (car L) L1) (cons (cadr L) L2)))))))

(define (fusion LT1 LT2 r?)
  (cond ((null? LT1) LT2)
        ((null? LT2) LT1)
        ((r? (car LT1) (car LT2)) (cons (car LT1) (fusion (cdr LT1) LT2 r?)))
        (else (cons (car LT2) (fusion LT1 (cdr LT2) r?)))))

(define (tri-fusion L r?)
  (if (or (null? L) (null? (cdr L)))
      L
      (k-decouper L (lambda (L1 L2) (fusion (tri-fusion L1 r?) (tri-fusion L2 r?) r?)))))
```

Pour ne pas prendre le mauvais réflexe de mettre apply partout une fois appris, quelques comparaisons :

```
(define ($andMap p? L)
  (cond ((null? L) #t) ; 3
        ((p? (car L)) ($andMap p? (cdr L)))
        (else #f)))
```

```
(define ($orMap p? L)
  (cond ((null? L) #f)
        ((null? (cdr L)) (if (p? (cdr L)) #t #f))
        ((p? (car L)) #t)
        (else ($orMap p? (cdr L)))))
```

```
; versions bourrins, qui vont jusqu'au bout hélas... :
(define ($orBourrin p? L)  ((not zero?) (apply + (map (lambda (x) (if (p? x) 1 0)) L))))
(define ($andBourrin p? L) (zero? (apply + (map (lambda (x) (if (p? x) 0 1)) L))))
```

III. Programmation impérative

La programmation impérative, dans sa philosophie, est très différente de ce que nous avons vu jusque là :

- En programmation impérative, on modifie explicitement la mémoire de la machine. On a des instructions exécutées en séquence et destinées à la modification de variables. L'itération est privilégiée, la récursivité encore peu optimisée. Les preuves mathématiques sont très difficiles mais l'exécution efficace (Pascal, C, Java).
- La programmation fonctionnelle procède par composition de fonctions récursives. La modification de l'état de la machine est implicite, gérée par le compilateur. On optimise la récursivité et fournit l'itération comme cas particulier. Les preuves en sont facilitées, mais l'exécution est plus lente…

Quand on parle de la famille de la « programmation applicative[4] », c'est celle où la seule structure de contrôle est la fonction au sens mathématique. Scheme en fait bien entendu partie !

Pour séquencer des instructions, on utilise (begin e_1 … e_n) comme vu précédemment.
Pour faire une affectation, on utilise (set! <symbole> <expression>). Il s'agit d'une des nouvelles formes spéciales de Scheme, car <symbole> n'est pas évalué. On parle d'opération de « mutation » : on fait muter la valeur d'une variable sans espoir de retour.

set! commence par rechercher la première liaison du symbole dans un des dictionnaires ; s'il n'y a pas de liaison, on a une erreur. On évalue l'expression à affecter dans l'environnement, et on remplace la liaison du dictionnaire par le résultat de l'expression. Au final, il n'y a donc pas de résultat.

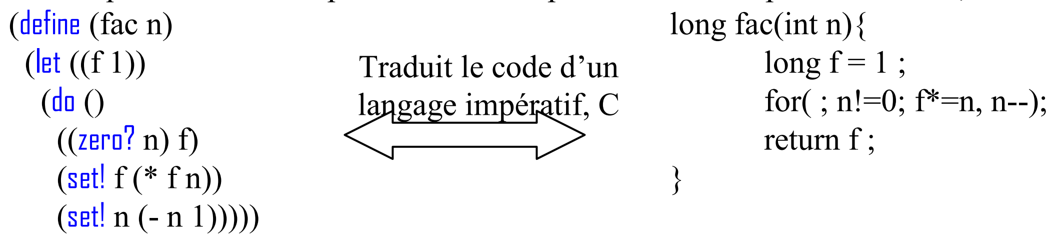
```
⌂ (define x 1)          ; on définit une variable x au top-level
⌂ (define y 2)          ; et une variable y
⌂ (let ((z x) (x 3))    ; on fait un nouveau dictionnaire. Toutes les variables montent en même temps !
    (set! x (+ x 10))   ; on va rechercher le x dans les dictionnaires en remontant. On trouve celui du let
    (set! y (+ y 20))   ; on recherche le y en remontant, on trouve celui du top-level
    (set! z (+ z 1))    ; le z était une copie du x défini au top-level, c'est la règle dans un let...
    (list x y z))       ; ((+ 3 10) (+ 2 20) (+ 1 1))
→ (13 22 2)
⌂ (list x y)            ; maintenant, on demande à voir le x et le y défini au top-level
→ (1 22)                ; on a jamais touché au x défini au top-level. Par contre on a modifié le y.
```

C'est toujours l'histoire des variables globales et locales. Les variables du top-level sont des variables globales, et après on définit des variables locales ; elles prennent naissance au début de leur bloc, et disparaissent à la fin. Pendant ce temps, elles masquent les variables des niveaux supérieurs.

```
⌂ (define (push x L)    ; on définit une fonction qui prend une liste L et un objet x
   (set! L (cons x L))  ; elle fait muter la liste en y rajoutant l'objet (donc elle empile)
   L)                   ; et elle nous rend la liste
⌂ (define L '(a b c))   ; définissons une liste pour tester tout ça...
⌂ (push 1 L)            ; on fait un push dedans
→ (1 a b c)             ; on nous rend le résultat de l'action, c'est bien (1 a b c)
⌂  L
→ (a b  c)              ; et pourtant a b c n'a pas bougé... pourquoi ?
```

Ici, on est dans le cas d'une procédure. Le passage des paramètres, comme dans quasiment tous les langages, se fait en « appel par valeur » (*call by value*) : on peut modifier un paramètre dans le corps de la procédure, mais sa valeur en sortie sera restaurée à celle qu'il avait en entrée !
Cependant, lorsque le paramètre d'une procédure est un pointeur (liste, vecteur, string), alors on peut modifier l'intérieur de l'objet de manière définitive dans la procédure, mais pas l'adresse de son point d'entrée. On a une référence directe sur la mémoire : ce qu'on fait y sera inscrit de manière définitive ; par contre, l'objet est toujours localisé au même endroit de la mémoire.

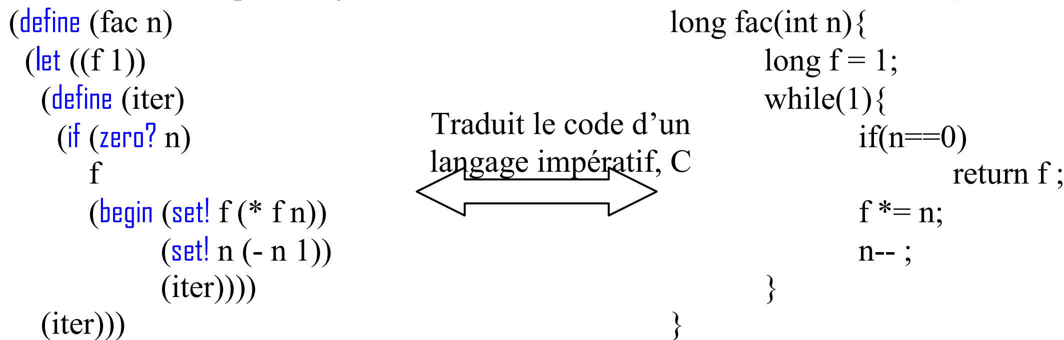
On dit qu'une boucle do pourvue d'un corps est un « do impératif ». Ainsi, la factorielle impérative est :




```
(define (fac n)
 (let ((f 1))
  (do ()
   ((zero? n) f)
   (set! f (* f n))
   (set! n (- n 1)))))
```

```
long fac(int n){
      long f = 1 ;
      for( ; n!=0; f*=n, n--);
      return f ;
}
```

Bien entendu, on peut toujours le faire sans utiliser do, avec le (define iter) habituel :




```
(define (fac n)
 (let ((f 1))
  (define (iter)
   (if (zero? n)
      f
      (begin (set! f (* f n))
             (set! n (- n 1))
             (iter))))
  (iter)))
```

```
long fac(int n){
      long f = 1;
      while(1){
            if(n==0)
                  return f ;
            f *= n;
            n-- ;
      }
}
```

Pour compter la somme des entiers impairs de 1 à 100, ou inverser une liste, avec un do impératif :

```
(let ((acc 0))
 (do ((i 0 (+ i 1)))
  ((> i 100) acc)
  (set! acc (+ acc (sqr i)))))
```

```
(define ($reverse L)
 (let ((acc '()))
  (do ()
   ((null? L) acc)
   (set! acc (cons (car L) acc))
   (set! L (cdr L)))))
```

IV. Valeur d'une λ-expression et fermeture

La valeur d'une λ-expression dans un environnement E est une fermeture : un couple <L, E> formé du texte L de la lambda et d'un pointeur sur E. C'est dans l'environnement de compilation et non dans l'environnement d'exécution que seront cherchées les valeurs des variables libres d'une λ.
En bref, la λ se souvient de l'environnement dans lequel elle a été compilée.

```
⌂ (define a 10)            ; on définit 'a' à 10 au top-level
⌂ (define foo
   (let ((a 1000))         ; on pose un environnement
     (lambda (x) (+ x a))))  ; la variable 'a' sera cherchée dans l'environnement courant, où a = 1000
⌂ (set! a (+ a 2))         ; on modifie la valeur de 'a' au top-level, donc maintenant il vaut 12
⌂ (foo a)                  ; la λ utilise son environnement, et elle fait donc (+ 12 1000)
→ 1012
```

Dans notre exemple, on a une λ faite d'un texte L = (+ x a) avec un pointeur sur E = (let ((a 1000)). Les valeurs des variables libres + et 'a' de la λ vont être cherchées dans cet environnement de compilation E. Au final, seules les variables paramètres (x) sont liées et donc mathématiquement saines.

V. Construction de générateurs

On veut programmer un générateur de nombres pairs, i.e. un objet qui nous délivre le nombre pair suivant à chaque fois qu'on l'appelle.
(gpair) → 0
(gpair) → 2, etc. etc.

Il nous faut donc un compteur n, mais on ne peut pas le mettre en variable globale. On va alors utiliser l'environnement de compilation d'une fermeture pour maintenir des variables privées.

```
(define gpair
  (let ((n -2))
    (lambda ()
      (set! n (+ n 2))
      n)))
```

D'abord on fait l'environnement, dans lequel on met les variables privées

Puis on fait la fonction. Ici, elle incrémente le compteur de 2 à chaque fois. Comme sa variable part de -2, elle commence en donnant 0.

On veut maintenant pouvoir obtenir des générateurs, chaque disposant d'un compteur n privé. Au lieu de nous rendre un nombre, on veut une fonction, c'est-à-dire une λ de tout ceci. Rien de plus simple :

```
(define (make-gpair)              ; avec des parenthèses, on signifie que tout ceci est maintenant une lambda
  (let ((n -2))
    (lambda ()
      (set! n (+ n 2))
      n)))
```

On peut ainsi définir plusieurs générateurs, chacun étant une application de la λ :

```
⌂ (define gen1 (make-gpair)
⌂ (define gen2 (make-gpair)
⌂ (begin (gen1) (gen1) (gen2) (gen2) (gen1) 'ok)
→ ok
⌂ (list (gen1) (gen2))
→ (6 4)
```

Sur le même principe, construisons un générateur de factoriels :

```
(define (gen-fac)
  (let ((n 0) (courant 1)) ; on calcule (n+1)! = (n+1)*n! où n! est stocké dans la variable privée 'courant'
    (lambda ()
      (set! n (+ n 1))
      (set! courant (* n courant))
      courant)))
```

Pour vérifier d'une part que cela marche, et d'autre part qu'il y a bien indépendance des générateurs, on fait une procédure de test. On veut (1! 2! … 11! 1! 2! 3!), en utilisant deux générateurs :

```
(define affiche-factoriels
  (lambda ()                              ; juste pour écrire en style Indiana…
    (let ((gen1 (gen-fac)) (gen2 (gen-fac)))  ; on se donne deux générateurs
      (define (iter n)                    ; on va utiliser le premier jusqu'à 11!
        (if (equal? n 11)
            (list (gen2) (gen2) (gen2))   ; et on termine en utilisant le second trois fois d'affilé
            (cons (gen1) (iter (+ n 1)))))
      (iter 0))))
```

On obtient bien (1 2 6 24 120 720 5040 40320 362880 3628800 39916800 1 2 6), générateur valide.
De même on construit un générateur de nombres de Fibonacci et une procédure de test :

```
(define (gen-fib)
  (let ((n1 1) (n2 1))
    (lambda ()
      (let ((tmp n2))
        (set! n2 (+ n1 n2))
        (set! n1 tmp)
        n2))))
```

```
(define affiche-fibonacci
  (lambda ()
    (let ((generateur make-gen-fib))
      (define (iter n)
        (if (equal? n 11)
            '()
            (cons (generateur) (iter (+ n 1)))))
      (iter 0))))
```

VI. Les mémo-fonctions, vers une programmation dans le style objet

On vient de voir qu'une fonction (une fermeture) dispose d'une mémoire privée. On peut considérer que celle-ci est constituée de neurones conservant l'état du système. Si on fait souvent le même calcul, on peut regarder dans les neurones si il a déjà été fait avant ; si c'est le cas, le résultat est déjà connu ! Cette idée classique en I.A. est devenue une stratégie algorithmique : programmation dynamique.

On programme des fonctions qui se souviennent des calculs déjà effectués : ce sont des fonctions à mémoire, soit des mémo-fonctions.

La fonction de fibonacci est de complexité exponentielle car elle passe son temps à refaire les calculs :
$f_{30} = f_{29} + f_{28} = (f_{28} + f_{27}) + f_{28} = 2.f_{28} + f_{27} = 2.f_{28} + f_{26} + f_{25} = 2.f_{28} + 2.f_{25} + f_{24} = \ldots$
On va utiliser des neurones pour stocker les calculs déjà faits, en les mettant sous forme de A-liste :

```
(define memo-fib
 (let ((AL '((0 0) (1 1))))              ; les neurones initiaux
  (define (calcul n)
   (let ((essai (assoc n AL)))           ; on regarde si le calcul a déjà été fait.
    (if essai
      (cadr essai)                                    ; si oui, alors on le sort de l'association
      (let ((res (+ (calcul (- n 1)) (calcul (- n 2)))))  ; sinon, on le fait…
       (set! AL (cons (list n res) AL))               ; …on le stocke pour ne plus le refaire…
       res))))                                        ;… et on peut renvoyer le résultat
  calcul))
```

Ainsi, on arrive à une fonction de Fibonacci en complexité linéaire au lieu de exponentielle !

On a cependant des problèmes à cause des variables privées. Si on veut par exemple réinitialiser un générateur, on ne peut pas. Il est impossible de modifier de force la mémoire privée d'une fermeture…
La stratégie est alors de demander à la fonction, depuis l'extérieur, de modifier son état local. La métaphore est un micro-monde d'acteurs, qui communiquent en s'envoyant des messages. Chacun devra :

- disposer d'un savoir-faire (avoir des méthodes)
- réagir à la réception d'un message (lancer la méthode qu'on lui demande)

Faisons le sur le générateur de nombre pair, et comparons à une classe Java :

```
(define (acteur-gpair)
 (let ((cpt '?))                  ; un neurone privé
  (define (this selecteur . args)
   (case selecteur
    ((reset!) (this 'init! -2))       ; l'auto-référence !
    ((init!) (set! cpt (car args)))
    ((get-cpt) cpt)
    ((next) (set! cpt (+ cpt 2)) cpt)
    (else (error "Méthode inconnue" selecteur))))
  (this 'reset!)
  this)) ; procédure qui dispatche selon le sélecteur
```

```
public class Gpair {
    private int cpt;
    public Gpair() {
        cpt = -2;
    }
    public int getCpt() {
        return cpt;
    }
    public int next() {
        cpt += 2;
        return cpt;
    }
    public void init(int cpt) {
        this.cpt = cpt;
    }
    public void reset() {
        this.init(-2);
    }
}
```

Le choix du mot « this » fait référence à Java : il s'agit de l'objet courant. On vient ainsi de définir 'this' en Scheme…

On peut continuer sur cette philosophie en faisant un acteur (pour ne pas dire un objet…) qui représente une pile. Il y aura donc une variable privée de type liste, et une réaction pour les messages 'empiler', 'depiler', 'vide?' et 'sommet'.

```scheme
(define (acteur-pile)
 (let ((L '()))                                          ; au départ, la pile est une liste vide
  (define (this selecteur . args)
   (case selecteur
    ((vide?) (null? L))                                  ; elle est vide : la liste est elle vide ?
    ((sommet) (if (null? L) (error "Pile vide !") (car L)))
    ((depiler) (if (null? L) (error "Pile vide !") (set! L (cdr L))))   ; on modifie la variable
    ((empiler) (set! L (cons (car args) L)))
    (else (error "Methode inconnue" selecteur))))
  this))

(define (make-memo-bin)                  ; on propose ici un acteur pour calculer le coefficient binomial
 (let ((neurones '()))
  (define (this selecteur . args)
   (case selecteur
    ((table) neurones)                   ; retourne la table de mémorisation
    ((calc?) (not (not (assoc args neurones))))        ; est-ce qu'un couple a déjà été calculé ?
    ((calc)
     (let ((n (car args)) (p (cadr args)))
      (if (or (= p 0) (= p n))                         ; si on est sur les bordures du triangle de pascal, 1
          1
          (let ((essai (assoc args neurones)))         ; sinon, on tente de prendre les résultats du calcul
           (if essai
               (cadr essai)                            ; si le calcul a déjà été fait, on le retourne
               (let ((res (+ (this 'calc (- n 1) p) (this 'calc (- n 1) (- p 1)) )))   ; et sinon… on le fait
                 (set! neurones (cons (list args res) neurones))                 ; on le mémorise
                 res))))))                                                       ; et on le renvoie
      ((reset) (set! neurones '() ))     ; on écrase la mémoire
      (else (error "Connais pas cette instruction etrange venue de loin..." selecteur))))
    this))

(define (make-circular-list L)           ; on propose ici un acteur prenant une liste et la rendant circulaire
 (let ((pointeur L))                     ; circulaire veut tout simplement dire de faire boucler le pointeur
  (define (this selecteur . args)
   (case selecteur
    ((type) "une liste circulaire")      ; retourne simplement une string
    ((valeur) (car pointeur))            ; quelle est la valeur actuelle ? je regarde le pointeur
    ((next)                                      ; je veux l'élément courant et avancer le pointeur
     (let ((elCourant (this 'valeur)))           ; je sauvegarde l'élément courant
      (if (null? (cdr pointeur))                         ; le pointeur est-il arrivé au bout de la liste ?
          (set! pointeur L)                              ; si oui, je le remet au début
          (set! pointeur (cdr pointeur)))                ; sinon je continue de l'avancer
      elCourant))                                ; et je rend l'élément courant que j'ai sauvegardé
    ((reset) (set! pointeur L))          ; on ré-initialise le pointeur de liste à l'origine
    ((init) (set! L (car args))          ; on initialise la liste circulaire interne à celle passée en argument
        (this 'reset))
    (else (error "Je ne comprend rien, inconnu, etc." selecteur))))
  this))
```

Testons l'acteur pour rendre une liste circulaire au top-level :

```scheme
⌂ (define roue (make-circular-list (list 'a 'b 'c 'd)))
⌂ (do ((i 1 (+ i 1)))   ((= i 10) (printf "\n"))  (printf "~a " (roue 'next))   ) → a b c d a b c d a
```

VII. Tableaux, ou vecteurs

Pour illustrer les tableaux (c.f. page 81 pour les primitives), faisons l'algorithme du drapeau hollandais. On a un tableau à trois couleurs (0 1 2), on souhaite le trier. On va se contenter de convertir un code Java.

```
public static void drapeauHollandais (int[] v) {      // en Java
    int b = 0, i = 0, r = v.length, tmp;
    // invariant : bleu[0,b-1], blanc[b,i-1], ?[i,r-1] et rouge[r,n-1]
    while (i < r) {
      switch (v[i]) {
        case 0:
          echanger(v,b,i);
          ++b; ++i;
          break;
        case 1:
          ++i;
          break;
        case 2:
          --r;
          echanger(v,r,i);
          break;
      }
    }
  }
```

```
public static void echanger (int[] v, int i, int j) {
    int tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
  }
```

```
(define (echanger! v i j)           ; v[i] ↔ v[j]
  (let ((tmp (vector-ref v i)))
    (vector-set! v i (vector-ref v j))
    (vector-set! v j tmp)
    v))
```

```
(define (drapeau v)
  (let (  (b 0) (r (vector-length  v)))
    (define (iter parcours)
     (if (>= parcours r)
         v
         (case (vector-ref v parcours)
          ((0)
           (echanger! v b parcours)
           (set! b (+ b 1))
           (iter (+ parcours 1)))
          ((1)
           (iter (+ parcours 1)))
          ((2)
           (set! r (- r 1))
           (echanger! v r parcours)
           (iter parcours))
          (else (error "Ce n'est pas ce qu'on entend actuellement par un drapeau hollandais..")))))
    (iter 0)))
```

```
; un drapeau aléatoire de longueur n

(define (random-drapeau n)
  (let ((final (make-vector n)))
    (do ((boucle 0 (+ boucle 1)))
      ((= boucle n) final)
      (vector-set! final boucle (random 3)))
    final))
```

```
; définition de la primitive vector-fill!
; met x dans chaque composante du vecteur v
(define ($vector-fill! v x)
  (let ((n (vector-length v)))
    (do ((i 0 (+ i 1)))
      ((= i n) (void))
      (vector-set! v i x))))  ; v[i] = x
```

VIII. Définir des fonctions locales récursives par letrec

On rappelle que le define dans une fonction sert à définir des fonctions locales, et non des variables. Si on voulait définir une petite fonction locale avec un let, on pouvait aussi :

```
(define (foo a)
   (let ((cube (lambda (x) (* x x x))))
      (cube (+ a 1))))
```

Par contre, si la fonction à définir est récursive, on ne peut utiliser let :

```
⌂ (let ((fac  (lambda (n)
                  (if (= n 0) 1 (* n (fac (- n 1))))))
      (fac 5))
→ undefined identifier : fac
```

Il y a un problème de fermeture avec la variable libre 'fac' qu'on cherche au top-level. Explications :

Pour évaluer (let ((x1 e1) ...) expr1 expr2 ...) dans un environnement E :
1. on commence par construire un dictionnaire D = {<x1,?>, <x2,?>, ...} que l'on chaîne à E. Soit E' l'environnement étendu E' = D→E.
2. on évalue dans un ordre arbitraire les expressions e1, ... **dans l'environnement E**, pour obtenir des valeurs v1, ... que l'on installe à la place des valeurs bidons ? dans D.
3. on évalue (begin expr1 expr2 ...) dans l'environnement étendu E' pour obtenir une valeur v.
4. on détruit le dictionnaire D.
5. on rend la valeur w.

• Dans l'exemple précédent, la lambda est évaluée au toplevel pour produire une fermeture <L,Global>. Lors de l'application de fac, la variable libre fac de son corps sera cherchée dans l'environnement de compilation Global, où elle n'existe pas !!! Le point 2 pose problème...

On peut résoudre le problème par deux méthodes :

```
(let ()   ; méthode habituelle, un peu débutant
  (define (fac n)
     (if (= n 0) 1 (* n (fac (- n 1)))))
  (fac 5))
```

```
(letrec ((fac (lambda n) ;utilise le let récursif, classieux !
                  (if (= n 0) 1 (* n (fac (- n 1))))))
     (fac 5))
```

En résumé :

- une fonction locale non récursive peut-être construite avec let. Dès qu'elle est récursive, on utilise letrec ou un define
- Une suite de define internes équivaut à un letrec.
- On ne fait un define interne ou un letrec que pour une fonction.
- Les fonctions d'une suite de define internes ou d'un letrec peuvent s'invoquer les unes les autres, elles sont mutuellement récursives.

```
; ne garder d'une liste que son premier et son dernier élément
(letrec ((debut (lambda (L) (car L)))
         (fin (lambda (L) (if (or (null? L) (null? (cdr L)))
                              (car L)
                              (fin (cdr L)))))
         (extremite (lambda (L) (list (debut L) (fin L)))))
   (extremite '(a b c d e)))
```

IX. Modification impérative des pointeurs

Quand on fait un (define d (cons 1 (cons 2 3))), la variable ne contient que l'adresse du premier doublet et du chaînage. C'est un pointeur, ou une référence vers le chaînage. De même, (cdr d) contient l'adresse de la seconde cellule de liste.
Il ne faut donc pas confondre le pointeur et l'objet pointé. Rappelons que pour les différencier, il y a eq? et equal?. Avec eq? on demande « les pointeurs sont-ils égaux ? » et avec equal? on demande « les objets s'affichent-ils de la même manière ? ».

Jusqu'à présent on a eu une approche purement fonctionnelle sur les listes en recopiant partiellement. Par exemple, (append L_1 L_2) clonait toutes les cellules de L_1 et faisait pointer la suite sur L_2.
On peut avoir envie d'éviter la recopie et d'accrocher physiquement la tête de queue de L_1 à L_2. Autrement dit, on ne veut faire aucune allocation avec un cons mais des effets de bord. On utilise append!.

Pour notre approche impérative, il faut introduire des procédures de mutation[5] sur le car et le cdr. Les noms sont assez logiques : set-car! et set-cdr!, analogues au set! vu en page 84. Attention cependant :

- On peut provoquer des erreurs difficiles à détecter. Par exemple, si une partie de L_1 pointe sur L_2 et qu'on modifie L_2, alors on modifiera L_1 en conséquence, sans peut-être se souvenir qu'ils sont reliés…
- Un objet est mutable si, et seulement si, il a été alloué par des procédures comme cons ou list qui ont demandé des allocations mémoires. Certains compilateurs sont laxistes, comme DrScheme…

```
(define L (list 'a 'b 'c'd)) ; mutable
(set-car! L 'e)
```

```
(define L '(a b c d))      ; non mutable
(set-car! L 'e) ; L est une constante !
```

Faisons quelques exemples de cette programmation par modification physique des listes :

```
(define ($append! L1 L2)     ; L1 non vide
 (do ((ptr L1 (cdr ptr)))
   ((null? (cdr ptr)) (set-cdr! ptr L2) L1))) ; aucun appel à cons. Toujours Θ(L1) en temps mais 0 en espace !
```

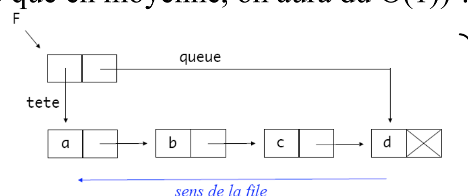
```
(define (list-set! L k x)   ; coût = O(k) en temps et 0 en espace
 (cond ((null? L) (error "Impossible !"))
       ((zero? k) (set-car! L x))
       (else (list-set! (cdr L) (- k 1) x)))) ; récursivité terminale. Une forme masquée d'itération…
```

```
(define (supprimer! L k)    ; k > 0
 (cond ((null? L) (error "Impossible !"))
       ((= k 1) (set-cdr! L (cddr L)))
       (else (supprimer! (cdr L) (- k 1)))))
```

```
(define (ins-apres! x y L)
  (cond ((null? L) (error "Element inexistant" y))
        ((equal? (car L) y) (set-cdr! L (cons x (cdr L))))
        (else (ins-apres! x y (cdr L)))))
```

On rappelle que, comme le passage des paramètres a lieu par valeur, le pointeur L sera restauré intact en sortie. Comme on ne peut pas modifié son adresse, on se contraint à k > 0. Si on veut éliminer cette contrainte, une solution simple consiste à faire une première boite « inutile » et ainsi on pourra opérer directement sur la suite. Par exemple, pour un polygone (liste de points), la cellule d'ancrage pourrait contenir sa longueur, mise à jour lors d'un ajout ou d'une suppression…

Faisons un exemple d'application avec une file d'attente. On veut que toutes les opérations soient en O(1). Ceci n'est pas réalisable en programmation purement fonctionnelle, même si on sait néanmoins obtenir du O(1) amorti (c'est-à-dire que en moyenne, on aura du O(1))[6].



Le principe est de maintenir un pointeur de tête et un pointeur de queue dans une cellule d'ancrage

MCours.com

```
(define (make-file-attente)      ; programmation d'une file d'attente par acteur (un peu paradigme objet…)
 (let ((F (cons '() '()))) ; F = < tête de la liste   ;   queue de la liste >
  (define (this selecteur . args)
   (case selecteur
    ((vide?) (null? (car F)))                 ; peu importe le (cdr F) si le car est vide !
    ((init!) (set-car! F '()))
    ((enfiler!)
     (let ((maBoite (cons (car args) '())))
      (if (this 'vide?)                        ; la liste était vide, on veut enfiler dedans
          (begin                               ; simple : tout le monde pointe sur ma boite, il n'y a qu'elle
           (set-car! F maBoite)
           (set-cdr! F maBoite))
          (begin                               ; par contre si elle n'était pas vide, alors je vais l'ajouter…
           (set-cdr! (cdr F) maBoite)          ; ce qui veut dire que le dernier élément de la liste pointe sur
           (set-cdr! F maBoite)))))            ; ma boite, et que la queue de liste pointe aussi dessus
    ((premier)
     (if (this 'vide?)
         (error "File vide !")
         (caar F)))
    ((defiler!)
     (let ((renvoyer (this 'premier))) ; comme d'habitude, mémoriser le résultat à renvoyer
      (set-car! F (cdar F)) ; on avait dans le car de F : a -> b -> c. On décide de pointer le prochain
      renvoyer))
    ((liste) (car F))          ; file->list
    (else (error "Methode inconnue" selecteur))))
  this))
```

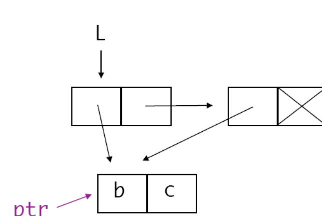
Un exemple plus amusant : définir la boîte (le cons) par un acteur :

```
(define (make-box . L)         ; retourne un acteur "boite". Attention au nombre d'arguments dans L !
 (let ((y (if (or (null? L) (null? (cdr L))) #f (cadr L)))      ; les deux champs x et y simulent car et cdr
       (x (if (null? L) #f (car L))))
  (define (this selecteur . args)      ; la définition de l'acteur courant
   (case selecteur
    ((toString) (format "box(~a . ~a)" x y))
    ((car) x)
    ((cdr) y)
    ((set-car!) (set! x (car args)))
    ((set-cdr!) (set! y (car args)))
    (else (error "Methode inconnue" selecteur))))
  this))
```

On s'attaque maintenant à un problème qui paraîtrait simple… On veut calculer le nombre de doublets d'un chaînage. Le premier réflexe peut-être simple : c'est un arbre, calculons le nombre d'éléments.

```
(define (nb-doublets expr)     ; algo faux !!!
 (if (not (pair? expr))
   0
   (+ 1 (nb-doublets (car expr))
        (nb-doublets (cdr expr)))))
```

```
(define L
  (let ((ptr (cons 'b 'c)))
    (cons ptr (cons ptr '()))))
```



Le problème, c'est qu'il peut y avoir un partage de mémoire : ce n'est plus un arbre.
La bonne solution va tourner autour de la mémoriser des doublets déjà visités…

On a besoin d'une fonction interne (aux expr LDV) retournant deux résultats :

- le nombre de doublets de expr non déjà vus
- une extension de LDV par les doublets de expr non déjà vus
  (on rajoute à la liste des visités ceux par qui on vient de passer, et on les compte)

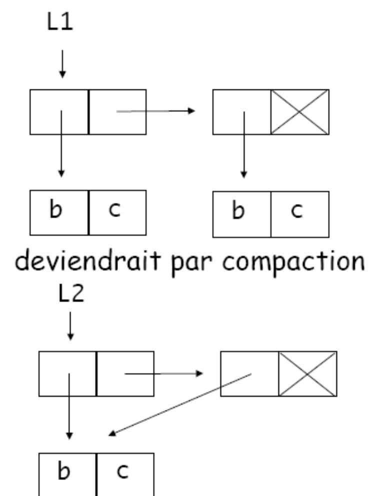
```
(define (nb-doublets expr)
 (define (aux expr LDV)    ; LDV : liste des doublets déjà visités, pour ne pas les compter à nouveau !
  (cond ((not (pair? expr)) (list 0 LDV))
        ((memq expr LDV)  ; memq est un analogue de member mais utilise une comparaison de pointeurs eq?
        (list 0 LDV))
        (else ; j'ai du doublet non déjà visité...
         (let* ((plongeeCAR (aux (car expr) (cons expr LDV)))
                (plongeeCDR (aux (cdr expr) (cons (car plongeeCAR) LDV)))
                (total (+ 1 (car plongeeCAR) (car plongeeCDR)))
                (LDV1 (list (cdr plongeeCAR) (cdr plongeeCDR))))
           (list total LDV1)))))
 (car (aux expr '())))
```

Une fonction très intéressante est la compaction du chaînage : on veut minimiser le nombre de doublets, donc en forçant des partages de mémoire. Comme avec l'algorithme précédent, il faudra maintenir une mémoire des doublets déjà compactés ; pour cela, on a fonction auxiliaire (aux expr mem) qui retourne :

- une version compactée de expr
- une extension de mem par les doublets de expr non déjà compactés
  (comme sur le principe de la précédente, elle fait le travail global et met à jour la mémoire)

L'algorithme est issu de Christian Queinnec, dans « Lisp, mode d'emploi » (Eyrolles 1984) :

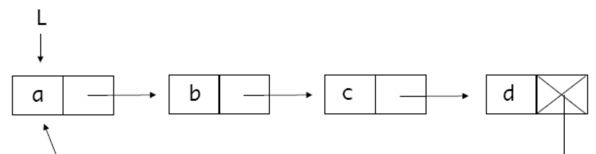
```
(define (compacter expr)          ; compactage d'un chaînage de doublets
 (define (aux expr mem)
  (if (not (pair? expr))
    (list expr mem)
    (let* ((cg (aux (car expr) mem))
           (cd (aux (cdr expr) (cadr cg)))
           (L (member expr (cadr cd))))
      (if L
        (list (car L) (cadr cd))
        (let ((R (cons (car cg) (car cd))))
          (list R (cons R (cadr cd))))))))
 (car (aux expr '())))
```



X. Structures circulaires

Un anneau est une liste rendue circulaire sur la dernière cellule, dont le cdr pointe en tête.

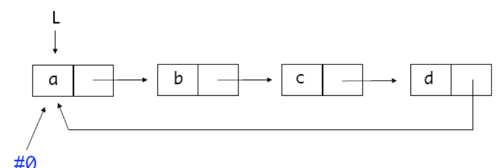
```
(define (rendre-circulaire! L) ; par effet de bord
  (append! L L) ; raccrocher la queue à la tête
  (void)) ; risque de boucler si affichage de L
```



La question intéressante est de savoir comment cet objet sera affiché. Il serait particulièrement sale que l'affichage se mette à boucler…

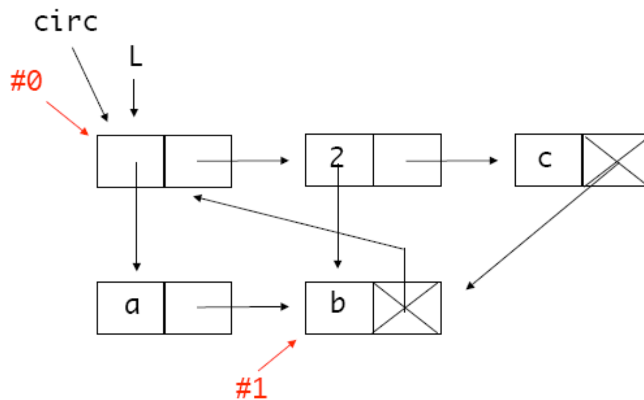
⌂ (define L (list 'a 'b 'c 'd))   ; *pour la mutation, on préfère utiliser list plutôt que la constante avec '*
⌂ (rendre-circulaire! L)
⌂ L
→ #0 = (a b c d . #0#)

L'affichage est intelligent : il détecte des points de partage, les défini et les utilise ! On peut afficher des graphes cycliques…



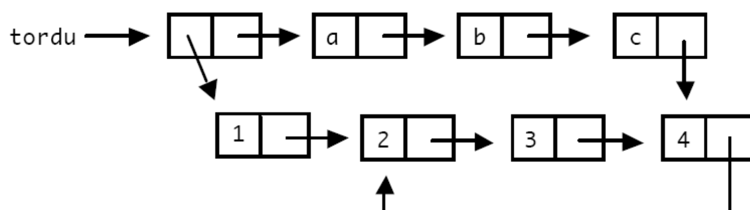
Les points de partage de DrScheme lui permettent d'afficher de façon compacte n'importe quel graphe cyclique. #i note un nouveau point de partage et #i# fait référence au point de partage déjà crée.

```
(define circ ; un graphe cyclique
  (let ((L (list (list 'a 'b) 2 'c)))
    (set-car! (cdr L) (cdar L))
    (set-cdr! (cdar L) L)
    (set-cdr! (cddr L) (cdar L))
    L))
```



Dès qu'un endroit est pointé par plus qu'un sommet, il y a définition d'un point de partage. Une fois les points définis, l'affichage les utilise simplement :
#0 = ((a . #1=(. #0#)) #1# c . #1#)

Entraînons nous maintenant à fabriquer un graphe d'après sa représentation, et à en deviner l'affichage. Quand on a une structure un peu compliquée, on ne va pas essayer de la faire en une fois. On peut la construire par petits morceaux…




```
(define cycle  ; construisons le cycle 2 3 4
 (let* ((ptr (cons 4 '())) ; on fait la boite du 4
       (ptr2 (cons 2 (cons 3 ptr)))) ; on fait le 2 3 4
  (set-cdr! ptr ptr2) ;et on fait boucler la boite du 4
  ptr2)) ; puis on renvoie l'ensemble
```

On a vu comment définir le cycle 2 3 4 et on a la boite du 4 bien à part, pour s'expédier dessus. D'où :

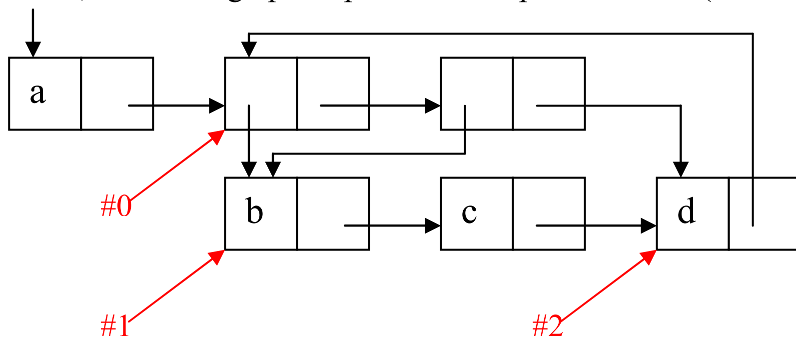
```
(define tordu
 (let* ((ptr (cons 4 '()))                      ; on fait la boite du 4 à part car il faudra la manipuler
        (ptr2 (cons 2 (cons 3 ptr)))            ; on fait le 2 3 4
        (ptr3 (cons 'a (cons 'b (cons 'c ptr)))) ; on fait a b c
        (final (cons (cons 1 ptr2) ptr3)))      ; on relie l'ensemble : le car est 1 2 3 4, le cdr est a b c
  (set-cdr! ptr ptr2)                           ; et on fait boucler la boite du 4
  final))
```

Il y a un point de partage sur le 2 et sur le 4, donc on défini #0 et #1. Soit #0 = (2 3 . #1#) et #1 = (4 #0#).
L'écriture du graphe ne pose plus alors de problèmes : ((1 . #0=(2 3 . #1=(4 . #0#))) a b c . #1#)
L'intérêt de cette écriture, outre d'être compacte et de ne pas boucler, est de pouvoir sauver un graphe sur le disque sans problème. Bien évidemment, ce n'est pas parce qu'un graphe boucle qu'on ne peut pas le sauver, il y a quantité de représentations ; mais ici, l'algorithme est fourni d'origine.

```
(call-with-output-file "tordu.dat"  ; exporter
 (lambda (p-out)  ; prendre le flux d'export
  (pretty-print tordu p-out)) ; expédier le graphe
 'replace) ; si le fichier existe déjà
```

```
(call-with-input-file "tordu.dat"   ; importer
 (lambda (p-in)            ; prendre le flux d'import
  (set! tordu (read p-in)))) ; lire et sauver la donnée
```

A l'inverse, trouver le graphe à partir de sa représentation : (a . #0=(#1 = (b c . #2 = (d . #0#)) #1# . #2#))



On lit a, on le fait tomber. On un point de partage #0 dont le car est un point de partage #1.
#1 est formé de b c et d'un point de partage #2, qui contient d et boucle sur un point de partage déjà défini : ok, on relie. On continue : on envoit sur les points de partage 1 et 2.

XI. Les entrées-sorties

En entrée, il y a des données que l'on envoie dans un « port d'entrée » pour qu'elle y soit traitée automatiquement. En sortie, c'est la donnée produire à la suite d'un traitement et envoyée sur le port de sortie. Les ports d'entrés peuvent être le clavier, un fichier, internet, une string… Dans les ports de sortie on a l'écran, le fichier, l'imprimante, le web et une string. En mode console on travaille clavier/écran.

En Scheme, les ports sont des objets de 1ère classe : on peut les passer en paramètres de fonctions.

```
(port? (current-output-port))        → #t
```

Les données reçues ou envoyées par un port sont sous la forme de suites de caractères ; c'est un flot. On prendra ici l'optique d'Emacs qui traite directement des flots d'objets Lisp. L'unité d'entrée-sortie ne sera pas le caractère car de trop bas niveau, mais une véritable expression Scheme !
Par exemple, un fichier contenant une liste de 5000 entiers ne contiendra en réalité qu'un seul objet Scheme, lisible par une seule instruction de lecture : on voit directement la liste. Ainsi, on pourra déposer un arbre de 2000 éléments avec une seule instruction d'écriture… comme une sérialisation en Java.

```
(define (doubler)
  (printf "Entrez un nombre ou une liste de nombres : ")
  (let ((x (read)))                      ; on lit une expression Scheme, aussi bien un nombre qu'une liste !
    (cond ((number? x) (* x 2))          ; est-ce qu'on a lu un nombre ? si oui, on le double
          ((list? x) (map (lambda (e) (* e 2)) x))    ; a-t-on lu une liste ? si oui, on double chaque élément
          (else (error "Entrée incorrecte !" x)))))        ; sinon… le type est imprévu

(define (valeur-moyenne)   ; on prend une fonction, des bornes et une précision, puis on donne la moyenne
  (define (integrale f a b h)
    (printf "Arguments reçus : ~a ~a ~a ~a\n" f a b h)
     (do ((pas a (+ pas (/ (- b a ) h)))  (total 0 (+ total (f pas)))) ; valeur moyenne de Id sur [0, 1] est ½
       ((>= pas b) (/ total h))))
  (printf "Donnez une expression en x : ")
  (let ((expression (read)))       ; l'objet lu est une expression Scheme complète, exemple (+ (* x x) 2)
    (printf "Borne gauche a = ")
    (let ((a (eval (read)))) ; on peut donner (/ (sqrt 7) 9). Il faut donc l'évaluer
      (printf "Borne droite b = ")
      (let ((b (eval (read))))
        (printf "Nombre de points d'interpolation n = ")
        (let ((n (eval (read))))
          (printf "Valeur moyenne de ~a est environ ~a\n"
           expression (integrale (eval (list 'lambda '(x) expression)) a b n)))))))
```

A partir de la lecture de l'expression, on construit une fonction. Ainsi, on a saisi une fonction au clavier !
Bien entendu, comme une fonction est également un objet Scheme, on aurait pu la saisir directement :

```
(define (valeur-en-0)
  (let ((f (eval (read))))   ; on saisie une fonction
    (f 0)))                  ; et on affiche son résultat en 0
```

On a donc trois procédures de lecture :

- (read-line) pour lire une suite de caractères et les assembler en chaîne, sans fin de ligne
- (read-from-string str) pour lire dans une chaîne de caractère. Attention, ça prendra le premier objet Scheme complet ! (read-from-string « 12 juin 1978 ») → 12
- (read-from-string-all str) pour donner la liste des objets à partir d'une chaîne de caractère

On a trois procédures normalisées pour l'écriture sur l'écran. (write x) affiche la valeur de x sous forme lisible par (read), (display x) l'affiche pour qu'on puisse la lire. Attention, printf n'est pas normalisé !

Concernant les différences entre les fonctions d'écritures, quelques petits exemples :

```
⌂ (write « a small string »)          ; on affiche en syntaxe Scheme, pour que read puisse lire
→ « a small string »
⌂ (display "a small string")          ; affiche pour un humain... mais ça fera 3 objets Scheme pour read
→ a small string
⌂ (write #\a)                         ; si on repasse avec read, on pourra bien lire un caractère
→ #\a
⌂ (display #\a)                       ; si on repasse avec read, on lira un symbole…
→ a
⌂ (write #(a b b b))
→ #4(a b)
⌂ (display #(a b b b))
→ #(a b b b)
```

```
(define (dichotomie)    ; déterminer le zéro d'une équation non nécessairement linéaire
  (printf "Entrez une fonction f : ")
  (let ((f (eval (read))))
    (printf "Borne a : ")
    (let ((a (eval (read))))
      (printf "Borne b : ")
      (let ((b (eval (read))))
        (printf "Nombre d'iterations : ")
        (let ((n (eval (read))))
          (chercheZero f a b n))))))
```

```
(define (chercheZero f a b n)
  (define (iter f a b n total)
    (let ((milieu (/ (+ a b) 2)))
      (cond ((> total n) milieu)
            ((> 0 (* (f a) (f milieu))) (iter f a milieu n (+ total 1)))
            (else (iter f milieu b n (+ total 1))))))
  (iter f a b n 0))
```

XII. Lecture et écriture dans un fichier-disque

Pour associer un port de sortie à un fichier sur le disque, on a deux choix :

- Soit ou ouvre et on ferme manuellement les ports
  (open-output-port f), de String → Port
  (close-output-port p), de Port → void
- On demande à Scheme de tout faire pour nous : (call-with-output-file chemin proc [mode])
  Le mode (optionnel) exprime de ce qui se passe si le fichier existe déjà : 'replace (écraser), 'append (ajouter), 'error (par défaut, on déclenche une erreur).

De même, pour lire un fichier on peut le faire manuellement ou automatiquement :

- (open-input-port f), (close-input-port f)
- (call-with-input-file f proc)

Si le fichier n'existe pas, c'est un cas d'erreur.

```
(define (creer-fichier-nombres f)        ;créer un fichier avec des nombres aléatoires, par exemple «foo.dat»
  (call-with-output-file f
    (lambda (p-out)
      (fprintf p-out "; fichier \"~a\"\n" f)
      (do ((i 0 (+ i 1)))
        ((= i 8) (void))
        (write (random 100) p-out)
        (display " " p-out))
      (newline p-out))
   'replace))
```

```
(define (somme-fichier f) ; sommer les nombres du fichier
  (call-with-input-file f
    (lambda (p-in)
      (define (iter nb somme)
        (let ((x (read p-in)))
          (if (eof-object? x)
              (printf "lu ~a nombres de somme ~a\n" nb somme)
              (iter (+ nb 1) (+ somme x)))))
      (iter 0 0))))
```

On rappelle que le texte des fonctions Scheme est vu comme des listes (pour la structure, des arbres). Si on veut compter le nombre de définitions dans un fichier, il suffit de regarder si l'objet sur lequel on passe est un doublet dont le premier élément est define. Ainsi :

```
(define (nb-definitions f)
 (call-with-input-file f
  (lambda (p-in)
   (define (iter nb)
    (let ((x (read p-in)))
     (cond ((eof-object? x) (printf "Le fichier contient ~a définitions !\n" nb))
           ((and (pair? x) (equal? (car x) 'define)) (iter (+ nb 1)))
           (else (iter nb)))))
   (iter 0))))
```

Notons qu'on préfère demander si l'objet est une pair, ce qui se fait en temps O(1), plutôt que si l'objet est une liste car c'est alors du O(n) : on devra la parcourir en entier pour regarder le dernier cdr…

```
(define (nb-lines f)            ; un analogue à la commande wc – l pour compter les lignes
 (call-with-input-file f        ; on ouvre le fichier...
  (lambda (p-in)                ; on réceptionne le flot
   (do ((str (read-line p-in) (read-line p-in)) (n 0 (+ n 1)))     ; on lit la LIGNE et on incrémente
    ((eof-object? str) n)))))                                      ; jusqu'à arriver en fin de fichier
```

On veut maintenant charger une fonction donnée à partir d'un fichier.

```
(define (load-only f fich)   ; f est un symbole, nom de fonction. Exemple (load-only 'fac « code1.scm »)
 (define (good-def? x)        ; est-ce que l'objet lu est bien ce qui nous intéresse ?
 (and (pair? x)               ; il faut que ce soit un doublet… (test en temps constant)
    (equal? (car x) 'define)  ; tel que son car soit le mot 'define
    (or (equal? (cadr x) 'f)  ; et qu'il définisse le fonction f, avec le style (define f (lambda (args) ...
       (and (pair? (cadr x)) (equal? (caadr x) f))))) ; ou avec le style (define (f args) ...
 (call-with-input-file fich
  (lambda (p-in)
   (define (iter x)
    (cond ((eof-object? x) (error "Fonction inexistante !"))
          ((good-def? x) (begin (display "La fonction a été chargée avec succès !") (eval x))) ; on la charge
          (else (iter (read p-in)))))
   (iter (read p-in)))))
```

Si on ne lit que des objets Scheme, alors l'interpréteur ne voit pas les commentaires ! Ainsi pour enlever les commentaires d'un fichier, il suffit de le lire, et de l'enregistrer. Au passage, on ne les aura pas vus…

```
(define (remove-comments f-in f-out)
 (call-with-input-file f-in              ; on lit donc le fichier
  (lambda (p-in)
   (display "Acces au fichier...")
   (call-with-output-file f-out          ; et on le sauvegarde
    (lambda (p-out)
     (define (iter objetCourant)         ; parcours du fichier et sauvegarde un à un des éléments…
      (if (eof-object? objetCourant)
          (newline p-out)
          (begin (pretty-print objetCourant p-out) (newline p-out) (iter (read p-in))))) ; mise en page correcte
     (iter (read p-in)))
   'replace))))
```

XIII. Utiliser le système d'exploitation

On dispose d'une librairie pour faire appel au système d'exploitation (require (lib « process.ss »)).
La procédure (system cmd) de la librairie permet de passer au système d'exploitation une chaîne contenant une commande du shell. Par exemple sous Unix, (system « ls – l »). Plutôt que de programmer le nombre de lignes, on aurait pu faire (system (format « wc –l ~a » f)) …

Utiliser system, quand on peut, est conseillé. Cependant il agit comme une procédure et renvoie #t ou #f !
Pour récupérer le résultat dans une chaîne, il faut rediriger la sortie courante vers un port associer à une chaîne ; autrement dit, considérer une chaîne comme un fichier.

```
(define (dir)
  (let ((p-out (open-output-string))
        (port (current-output-port)))
    (current-output-port p-out)
    (system "ls -l")
    (close-output-port p-out)
    (current-output-port port)
    (get-output-string p-out)))
```

On peut bien entendu faire des commandes différences selon le système d'exploitation qu'on aura détecté.

```
(if (file-exists? "code10.scm")
    (begin (remove-comments "code10.scm" "code10-nocomments.scm")
           (case (system-type)
             ((macosx)   (system "open code10-nocomments.scm"))
             ((windows) (system "type code10-nocomments.scm"))
             ((unix)       (system "cat code10-nocomments.scm")))))
```

XIV. Aller chercher des informations automatiquement sur Internet

On va utiliser la fonction tcp-connect, qui se connecte à un serveur Web par le port HTTP (80). Pour dialoguer avec un serveur par ce port, il faudra s'en tenir au protocole HTTP ! Connectons nous :

```
(define-values (p-in p-out) ; on définit deux choses à la fois. Ce sont bien 2 valeurs et pas une liste…
  (tcp-connect « www.nineplanets.org » 80)))
```

Cette fonction nous retourne les ports d'entrée-sortie , p-in et p-out, qui sont des <tcp-[in/out]put-port>.
Pour accéder à une page, il faut émettre une requête GET dans la syntaxe du protocole HTTP 1.0.
Une requête peut prendre plusieurs lignes, mais doit se terminer par une ligne vide. Ainsi :

```
(fprintf p-out « GET http://www.nineplanets.org/mars.html HTTP/1.0\n\n")
```

Le serveur Web prépare la page pour p-in... mais ne l'envoie pas. C'est à nous de lire le fichier peu à peu.

- Il suffit de faire des (read-line p-in) jusqu'à #<eof> pour lire le fichier ligne à ligne !
- Attention, la page Web commence par un **header** [en-tête] de plusieurs lignes renseignant le contenu actuel du fichier, qui débute après une ligne vide :

```
> (do ((i 0 (+ i 1)))
    ((= i 11) (void))
    (printf "~a" (read-line p-in)))
HTTP/1.1 200 OK
Date: Sat, 09 Apr 2005 14:19:01 GMT
Server: Apache/1.3.33 (Unix) mod_fastcgi/2.4.2 FrontPage/5.0.2
Last-Modified: Thu, 07 Apr 2005 21:05:28 GMT
ETag: "6c2ce8-7c72-4255a098"
Accept-Ranges: bytes
Content-Length: 31858
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
......
```

Une fois qu'on a accès à la page HTML et qu'on sait la parcourir, il faut y puiser des informations. Pour cela, on utilise des expressions régulières (regexp) ; elles sont analogues à celles utilisées sous Unix par egrep. Il s'agit d'un motif qui décrit une chaîne, avec des opérateurs pour combiner des regexp plus petites. Voici quelques exemples :

· Présence d'un caractère dans une chaîne

⌂ (regexp-match « c » « abcdec »)
→ ("c")

⌂ (regexp-match « p » « abcdec »)
→ #f

· Les caractères spéciaux

⌂ (regexp-match « . » « abcdec ») *; on cherche n'importe quel caractère*
→ ("a")
⌂ (regexp-match « [.] » « abcdec ») *; on cherche le caractère .*
→ ("a")

· Obtenir toutes les solutions

⌂ (regexp-match* « . » « abcdec »)
→ ("a" "b" "c" "d" "e" "c")

· Présence d'une sous-chaîne

⌂ (regexp-match « [.]jpg » « images foo.jpg et bar.jpg »)
→ (".jpg")

· Intervalles

⌂ (regexp-match « [a-z] » « THX-sound 2405 ») *; on veut une lettre minuscule*
→ ("s")
⌂ (regexp-match « [^A-Z] » « THX-sound 2405 ») *; on veut ce qui n'est pas une lettre majuscule*
→ ("-")
⌂ (regexp-match « [^a-zA-Z-] » « THX-sound 2405 ») *; ni minuscule, ni majuscule, ni tiret*
→ (" ") *; … la première chose rencontrée est l'espace*

· Séquences

⌂ (regexp-match « [a-z]+ » « THX-sound 2405 ») *; une séquence de une minuscule ou plus*
→ ("sound")
⌂ (regexp-match « [a-zA-Z]*[.]jpg » « image foo.jpg hop ») *; 0 ou plusieurs lettres, suivis de .jpg*
→ ("foo.jpg")
⌂ (regexp-match* « [a-zA-Z]*[.]jpg » « images foo.jpg et bar.jpg »)
→ ("foo.jpg" « bar.jpg »)

· Conditions multiples (disjonction)

⌂ (regexp-match « 0|1 » « 45b1d08 ») *; on cherche un 0 ou un 1*
→ ("1")

· Position particulière

⌂ (regexp-match « ^ab » « abcde ») *; ^ en tête pour un début de chaîne*
→ ("ab")
⌂ (regexp-match « ab$ » « abcde ») *; $ en queue pour une fin de chaîne*
→ #f

· Présence facultative

⌂ (regexp-match « colou?r » « color ») *; on veut colour ou color. Le caractère u est facultatif*
→ ("color")

La primitive (regexp-replace str1 str str2) permet de remplacer une sous-chaîne str1 de str par str2 :
(regexp-replace « gif|jpg » « lion.gif et jaguar.jpg » « bmp ») → « lion.bmp et jaguar.jpg »
Si on met des sous-regexp entre parenthèses, cela les numérote et permet de s'en resservir :
(regexp-replace « ([a-zA-Z]+) ([0-9]+) » « en janvier 2002 je dois » « \\2 (\\1) »)
→ « en 2002 (janvier) je dois »

• Allez, un exemple... Cherchons la taille en octets de la page HTML sur laquelle on s'est connecté plus haut [pages 18-19]. On peut le savoir avant de télécharger complètement la page puisque c'est dans le *header* : il suffit de chercher la ligne contenant Content-Length

```
(define-values (p-in p-out)
  (tcp-connect "www.nineplanets.org" 80))

(fprintf p-out "GET http://www.nineplanets.org/mars.html HTTP/1.0\n\n")

(define the-good-line
  (do ((str (read-line p-in) (read-line p-in)))
    ((regexp-match "Content-Length" str) str)))

(close-input-port p-in)
(close-output-port p-out)

(printf "Cette page HTML contient ~a octets !\n"
        (car (regexp-match "[0-9]+" the-good-line)))
```

```
Cette page HTML contient 31858 octets !
```

Stratégie générale d'extraction d'information sur internet :

- On se connecte en TCP
- On prépare la page avec (fprintf p-out « GET …
- On fait une première passe d'expression régulière pour trouver la ligne dont on sait qu'elle a le motif intéressant.
- On fait une seconde passe pour extraire la valeur qui nous intéresse.

On veut par exemple connaître la valeur du Nasdaq. On commence par repérer un site de bourse et on se connecte à sa page : http://www.boursorama.com/cours.phtml?symbole=%24COMPX
On identifie dans le code les lignes qui nous intéressent

```
<TD  NOWRAP align="left"> <A target="_top">Nasdaq Comp</A>
</TD>
<TD  NOWRAP align="right">2 336.74(c)</TD>
<TD  NOWRAP align="right"><span class="vardown">-0.13%</span></TD>
```

Et maintenant, on va extraire automatiquement la valeur et la variation du Nasdaq :

```
(define (nasdaq)
  (define-values (p-in p-out)                          ; on se connecte puis on prend la page
    (tcp-connect "www.boursorama.com" 80))
  (fprintf p-out "GET http://www.boursorama.com/cours.phtml?symbole=%24COMPX HTTP/1.0\n\n")
  (let ((the-good-line (do ((str (read-line p-in) (read-line p-in)))      ; on se place au bon endroit
                         ((regexp-match "Nasdaq Comp" str) str)))
        (valeur (begin (read-line p-in) (read-line p-in)))          ; deux lignes après il y a la valeur
        (variation (read-line p-in)))                               ; une ligne après la variation
    (close-input-port p-in)             ; on ferme tout ça...
    (close-output-port p-out)
    (let ((nasdaqVal (car (regexp-match "[0-9][0-9 .]+" valeur)))  ; extraction. On veut le car du résultat
          (nasdaqVariation (car (regexp-match "[0-9][0-9 .]+" variation))))
      (list 'Valeur nasdaqVal 'Variation (string->number nasdaqVariation)))))
```

Les expressions régulières peuvent tout aussi bien nous servir pour extraire de l'informations des fichiers. Par exemple, on regarde comment une image GIF est construite[7]. On voit que les 6 premiers octets (char) contiennent le type de l'image : GIF87a ou GIF89a. Si on veut le type, il suffira donc d'afficher ces caractères.
Après le type, on a la largeur et la hauteur. On ne peut pas les afficher directement, il faut les changer de base. Par exemple, pour une largeur de n_1n_2 ce sera $n_1 + 256*n_2$ en base décimale !

Il n'y a plus qu'à ouvrir un port d'entrée sur un fichier gif, et de lire les caractères qu'il nous faut avec un read-char !

```scheme
;;; decodage du format d'une image gif
(define (size-of-gif-image fich)      ; fich est le nom d'un fichier xxxx.gif
  (let ((v (make-vector 4)))
    (call-with-input-file fich
      (lambda (p-in)
        (printf "L'image est de type ")     ; je lis les 6 premiers octets [type de l'image]
        (do ((i 0 (+ i 1)))
          ((= i 6) (newline))
          (display (read-char p-in)))           ; je stocke les 4 octets [char] suivants dans un vecteur
        (let ((v (build-vector 4 (lambda (i) (char->integer (read-char p-in))))))
          (printf "v = ~a\n" v)
          (printf "L'image a donc pour taille ~a x ~a\n"
                  (+ (vector-ref v 0) (* 256 (vector-ref v 1)))
                  (+ (vector-ref v 2) (* 256 (vector-ref v 3)))))))))
```

Quelques développements :

- On peut construire automatiquement la liste des sites qui répondent à une recherche donnée sur google. Tout d'abord, on regarde comment il poste ses requêtes en faisant un exemple. Ensuite, on se déplace dans les balises avec les expressions régulières. Si on met une itération sur les pages, on peut construire la liste de tous les sites en progressant dans les pages suivantes.

EXERCICES SUPPLEMENTAIRES

Exercice 11.7 **Le problème de Josephus** [Flavius[i]]. L'astucieux Josephus a décidé que sa troupe devait se suicider avant d'être capturée. Il proposa à ses 40 soldats – musclés mais faibles en algorithmique – la stratégie suivante : après s'être mis en cercle, il numérote les soldats de 1 à 40. Les soldats se suicident de 3 en 3 jusqu'au dernier. Vous avez deviné ? Il avait trouvé la bonne place pour être le dernier ☺…

Généralisons le problème à n soldats, tués de p en p, et notons J(n,p) le numéro du dernier et heureux soldat qui reste seul. On vous demande de programmer une fonction (josephus n p) retournant J(n,p). Exemples :

J(5,3) = 4, J(7,2) = 7, J(10,5) = 3

Stratégie : on part de la liste des entiers de 1 à n, que l'on transforme en *anneau*. On boucle ensuite tant que n > 1, en tuant un soldat tous les p soldats [on supprime physiquement un élément de la liste chaque fois]…

Une variante JL consiste à rendre la liste des numéros des soldats tués : JL(5,3) = (3 1 5 2 4).

N.B. Les mathématiciens savent des choses de mathématiciens sur les J(n,p)…

(http://mathworld.wolfram.com/JosephusProblem.html)

i

[1] Dans le livre « Structure and Interprations of Computer Programs », le magicien tient une boule de cristal dans laquelle sont inscrits eval/apply, qui sont les piliers de Scheme.

[2] Dans le langage APL, il y avait des one-liners : de grosses opérations réalisées en une seule ligne. APL est un langage cré par Ken Iverson dans les années 1960. Il s'agissait au départ d'une notation pour décrire des idées mathématiques, consistant en un ensemble de symbole et une syntaxe décrivant le traitement des données. Dans APL, il y a de nombreux opérateurs grecs et de très hauts niveaux, comme Δ pour le déterminant. En conséquence les programmes peuvent être très cours… et illisibles.

[3] raisonnement par l'absurde : le contraire ne peut être prouvé, donc...

[4] Voir la synthèse de Christian Queinnec et Pierre Weis, « Programmation Applicative : Etat des lieux et perspectives ».

[5] L'affectation est un cas particulier de mutation appliquée à des variables.

[6] F.W. Burton, « An efficient implementation of FIFO queues », Information Processing Letters, 1982, vol 14, pp 205-206

[7] http://www.colosseumbuilders.com/imageformats/gif87a.txt

MCours.com