# *Le langage Scheme*

*August 2022*

Ce document propose un tour d'horizon du langage Scheme. [1] Pour une introduction à la programmation avec le langage Scheme et des applications bioinformatiques, l'ouvrage de Laurent Bloch est excellent.[2] Pour un développement plus approfondi, se référer au livre de CHarazain.[3]

1. Traduction française d'un tutoriel de l'Université de Washington de 2003, disponible à l'adresse suivante : `https://courses.cs.washington.edu/courses/cse341/03wi/scheme/`.

## *Les bases de Scheme*

### *Caractéristiques du langage*

Parmi les caractéristiques principales du langage Scheme, on distingue les suivantes : dialecte de Lisp, essentiellement fonctionnel (mais pas seulement), typage dynamique sûr, stockaga basé sur la pile avec nettoyage automatique de la mémoire, passage par valeur avec une sémantique de pointeurs, portage lexical, fonctions de première classe et fonctions anonymes, syntaxe simple et régulière, peut fonctionner en mode interprété ou compilé.

Les domaines d'application de Lisp incluent l'intelligence artificielle (systèmes experts, planification, etc.), la simulation et la modélisation, la programmation d'applications (Emacs, CAD, Mathematica) et le prototypage rapide.

Lisp a été développé vers la fin des années 50 par John McCarthy. Le dialecte Scheme a été développé par Guy Steele et Gerry Sussman au milieu des années 70. Dans les années 80, le standard Common Lisp a été élaboré. Common Lisp est un langage à tout faire possédant de nombreuses fonctionnalités.

### *Structures de données basiques et types d'opération*

Voici quelques exemples de structures de données primitives :

— nombres : entiers (1, 4, -3, 0), réels (0.0, 3.5, 1.23E+10), rationnels (2/3, 5/2)

— symboles (fred, x, a12, `set!`)

— booléens : Scheme utilise les symboles spéciaux `#f` and `#t` pour représenter les valeurs faux et vrai

— chaînes de caractères (”hello sailor”)

— caractères (`#\c`)

La casse n'est généralement pas significative (sauf dans le cas des caractères ou des chaînes). Notons que les caractères +, - ou ! peuvent

se retrouver n'importe où dans les symboles, mais pas les signes de parenthèses. Voici quelques-uns des opérateurs de base que Scheme fournit pour les types discutés ci-dessus :

— opérateurs arithmétiques (`+`, `-`, `*`, `/`, `abs`, `sqrt`)

— opérateurs relationnels (`=`, `<`, `>`, `<=`, `>=`)

— opérateurs relationnels pour données arbitraires (`eqv?`, `equal?`)

— opérateurs logiques (`and`, `or`, `not`) : `and` et `or` sont évalués en court-circuit.

Certains opérateurs sont appelés *prédicats*, c'est-à-dire qu'ils agissent comme opérateurs de vérité. En Scheme, ils retournent `#f` ou `#t`. Une particularité à noter : sous MIT Scheme, une liste vide est équivalente à `#f`, et `#f` est affiché comme `()`. Un style recommandé est toutefois d'écrire `#f` ou `#t` pour représenter les valeurs vrai et faux, et `()` pour désigner une liste vide. Parmi les prédicats, on distingue les opérateurs relationnels cités plus haut ainsi que : `number?`, `interger?`, `pair?`, `symbol?`, `boolean?`, `string?`.

### *Utilisation d'opérateurs et de fonctions*

Scheme propose une syntaxe unifiée pour invoquer des fonctions :

```
(function arg1 arg2 ... argN)
```

Cela signifie que tous les opérateurs, incluant les opérateurs arithmétiques, possèdent une syntaxe préfixée. Les arguments sont passés par valeur (excépté dans le cas des *formes spéciales*, discutées plus loin, pour permettre des choses intéressantes telles que les court-circuits).

Voici quelques exemples :

```
(+ 2 3)
(abs -4)
(+ (* 2 3) 8)
(+ 3 4 5 1)
```

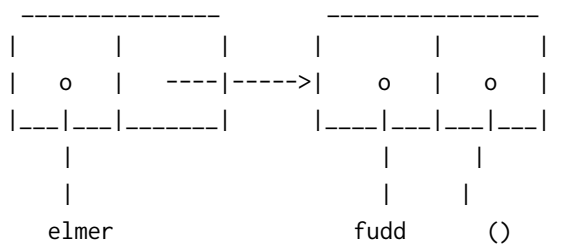
## *Le type de données liste*

Le type de données disponible de base le plus important en Scheme est la liste. Les listes sont illimitées, et consistent en une collection possiblement hétérogènes de données. Voici quelques exemples :

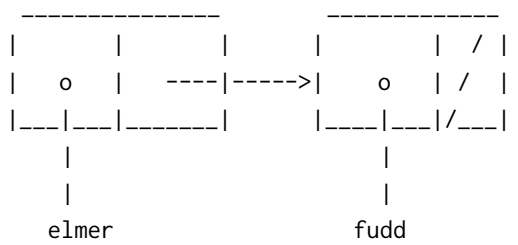
```
(x)
(elmer fudd)
(2 3 5 7 11)
```

```
(2 3 x y "zoo" 2.9)
()
```

Une représentation sous forme de boîtes fléchées des élements d'une liste est données ci-dessous :




Ou, de manière équivalente :




Remarques :

— `(x)` est différent de `x`

— `()` désigne la liste vide

— listes de listes : `((a b) (c d))` ou `((fred) ((x)))`

— Les listes en Scheme peuvent contenir des éléments de différents types : `(1 1.5 x (a) ((7)))`.

Voici quelques fonctions importantes qui opèrent sur les listes :

— `length` retourne la longueur d'une liste

— `equal?` teste si deux listes sont égales (récursivement)

— `car` renvoit le premier élément d'une liste

— `cdr` renvoit tout ce qui suit le premier élément d'une liste

— `cons` créé une nouvelle cellule de liste (une cellule de type `cons`)

— `list` construit une liste

Scheme offre également des compositions spécifiques de `car` et `cdr`. Par exemple, `(cadr s)` est défini comme `(car (cdr s))`.

Les prédicats pour les listes sont `null?` (la liste est-elle vide ?) et `pair?` (cet élément est-il une liste non vide ?).

### *Evaluer des expressions*

Les utilisateurs interagissent géénralement avec Scheme à l'aide d'un système appelé *read-eval-print-loop* (REPL). Scheme attend que l'utilisateur saisisse une expression, la lit, l'évalue, et affiche la valeur retournée. Les expressions Scheme (souvent appelée S-expressions, pour *Symbolic Expressions*) sont soit des lists soit des atomes. Les listes sont composées d'autres S-expressions (notons la définition récursive). Les listes sont souvent utilisées pour représenter des appels de fonction, où une liste correspond en un nom de fonction suivi de ses arguments. Toutefois, les listes peuvent également être utilisées pour représenter des collections arbitraires de données. Par la suite, on écrira généralement `<S-expression> => <return-value>` lorsque l'on souhaite montrer une S-expression et le résultat de son évaluation. Par exemple :

```
(+ 2 3)      => 5
(cons 1 () ) => (1)
```

Les règles d'évaluation sont les suivantes :

1. Les nombres, chaînes, `#f` et `#t` sotn des littéraux et sont auto-évalués.
2. Les symboles sont traités comme des variables, et pour les évaluer, leur liaison est recherchée dans l'environnement courant.
3. Pour les listes, le premier élement spécifie la focntion. Le reste des élements de la liste définissent les arguments. On évalue le premier élement dans l'environnement courant pour trouver la fonction, et on évalue chacun des arguments dans l'environnement avant d'appeler la foncton sur ces valeurs. Par exemple :

```
(+ 2 3)              => 5
(+ (* 3 3) 10)       => 19
(= 10 (+ 4 6))       =>  #t
```

### *Utilisation des symboles (atomes) et listes en tant que données*

Si l'on essaie d'évaluer `(list elmer fudd)`, on obtient une erreur. Pourquoi ? Parce que Scheme traite l'atome `elmer` comme un nom de variable et cherche sa liaison, qu'il ne trouve pas. Il est donc nécessaire de "quoter" [2] les noms `elmer` et `fudd`, ce qui implique que Scheme va les traiter de manière littérale. Scheme offre une syntaxe spécifique pour cela. La règle d'évaluation pour les objets quotés est que ceux-ci s'auto-évaluent.

2. On utilisera le terme anglais, *quote*, et ses déclinaisons anglicistes par souci de commodité.

```
'x                    => x
(list elmer fudd)     => error! elmer is unbound symbol
(list 'elmer 'fudd)   => (elmer fudd)
(elmer fudd)          => error! elmer is unknown function
'(elmer fudd)         => (elmer fudd)
(equal? (x) (x))      => error! x is unknown function
(equal? '(x) '(x))    => #t
(cons 'x '(y z))      => (x y z)
(cons 'x () )         => (x)
(car '(1 2 3))        => 1
(cdr (cons 1 '(2 3))) => (2 3)
```

Notons qu'il existe trois manières de créer une liste :

1. `'(x y z) => (x y z)`
2. `(cons 'x (cons 'y (cons 'z () ))) => (x y z)`
3. `(list 'x 'y 'z) => (x y z)`

De manière interne, les symboles quotés et les listes sont représentés à l'aide de la fonction spéciale `quote`. Lorsque le lecteur lit l'expression `'(a b)` il la traduit en `(quote (a b))`, et la transmet à l'évaluateur. Lorsque ce dernier soit une expression de la forme `(quote s-expr)`, il retourne seulement `s-expr` ; `quote` est appelé une "forme spéciale" car il n'y a pas d'évaluation de son argument, à la différence des autres opérations Scheme. La marque de quotation est un exemple de ce que l'on appelle "syntactic sugar". [3]

3. Alan Perlis : "syntactic sugar causes cancer of the semicolon".

```
'x          => x
(quote x)   => x
```

## *Variables*

Scheme dispose à la fois de variables locales et globales. En Scheme, une variable est un nom associé à un certain objet de données (à l'aide d'un pointeur). Il n'y a pas de déclaration de type pour les variables. La règle pour évaluer les symbole est la suivante : un symbole est évalué à la valeur de la variable qu'il nomme. On peut lier des variables à l'aide de la *forme spéciale* `define` :

```
(define symbol expression)
```

utiliser `define` revient à lier `symbol` (nom de variable) au résultat de l'évaluation de `expression`. On considère `define` comme une forme spéciale puisque le premier paramètre `symbol` n'est pas évalué.

La ligne suivante déclare une variable appelée `clam` (si elle n'existe pas déjà) et la fait référer à la valuer 17 :

```
(define clam 17)
clam          => 17

(define clam 23)  ; this rebinds clam to 23
(+ clam 1)    => 24

(define bert '(a b c))
(define ernie bert)
```

Scheme utilise des pointeurs : `bert` et `ernie` pointent à présent vers la même liste.

*Variables à portée lexicale avec* `let` *et* `let*`

On utilise la forme spéciale `let` pour déclarer et lier des variables temporaires, locales. La syntaxe est la suivante :

```
;; general form of let
(let ((name1 value1)
      (name2 value2)
  ...
      (nameN valueN))
   expression1
   expression2
   ...
   expressionQ)
```

Application :

```
;; reverse a list and double it

;; less efficient version:
(define (r2 x)
  (append (reverse x) (reverse x)))

;; more efficient version:
(define (r2 x)
  (let ((r (reverse x)))
(append r r)))
```

Le problème avec `let` est que tandis que les liaisons sont effectuées, les expressions ne peuvent faire référence aux liaisons réalisées précédemment. L'exemple suivant ne fonctionnera pas car `x` n'est pas connu en dehors du corps de l'expression :

```
(let ((x 3)
      (y (+ x 1)))
    (+ x y))
```

Pour résoudre ce problème, Scheme fournit la forme spéciale `let*` :

```
(let* ((x 3)
       (y (+ x 1)))
    (+ x y))
```

## *Définir ses propres fonctions*

### *Fonctions anonymes lambdas*

On utilise les formes spéciales `lambda` pour créer des fonctions anonymes. La forme spéciale s'écrit :

```
(lambda (param1 param2 ... paramk)  ; list of formals
        expr)                       ; body
```

L'expression `lambda` est évaluée en tant que fonction anonyme qui, lorsqu'elle est appelée, accepte `k` arguments et retourne le résultat de l'évaluation de `expr`. Comme attendu, les paramètres ont une portée lexicale et ne peuvent être utilisés que dans `expr`.

Exemple :

```
(lambda (x1 x2)
(* (- x1 x2) (- x1 x2)))
```

L'évaluation de l'exemple ci-dessus résulte en une fonction anonyme, avec laquelle on ne fait rien. Le résultat d'une expression `lambda` peut être appliqué directement en lui fournissant des argumebrs, comme dans l'exemple suivant, qui retourne la valeur 49 :

```
((lambda (x1 x2)
 (* (- x1 x2) (- x1 x2)))
 2 -5)   ; <--- note actuals here
```

### *Définir des fonctions nommées*

Si l'on prend la peine de définir des fonctions c'est souvent pour les réutiliser plus tard. Pour cela, on lie le résultat d'une `lambda` à une variable grâce à `define`, comme on le ferait avec n'importe quelle autre valeur. [4]

```
(define square-diff
(lambda (x1 x2)
(* (- x1 x2) (- x1 x2))))
```

Puisque la définition de fonctions est une tâche particulièrement commune, Scheme fournit une version simplifiée de `define` qui ne requiert pas l'usage de `lambda` :

4. Ceci illustre à quel point les fonctions sont des objets de première classe en Scheme. Cet usage de `define` n'est pas différent de la liaison entre des variables et d'autres types de données.

```
(define (function-name param1 param2 ... paramk)
        expr)
```

Voici quelques exemples d'utilisation de `define` dans ce contexte :

```
(define (double x)
(* 2 x))

(double 4) => 8

(define (centigrade-to-fahrenheit c)
(+ (* 1.8 c) 32.0))

(centigrade-to-fahrenheit 100.0) => 212.0
```

Les fonctions peuvent être sans argument :

```
(define (test) 3)
(test)  => 3
```

Notons que ce n'est pas la même chose que de lier une variable à une valeur :

```
(define not-a-function 3)
not-a-function    => 3
(not-a-function)  => ;The object 3 is not applicable.
```

### *Egalité et identité :* `equal?`, `eqv?`, `eq?`

Scheme fournit trois primitives pour les tests d'éqgalité ou d'identité :

1. `eq?` consiste en une comparison de pointeurs. La valeur de retour est `#t` si et seulement si les arguments réfèrent littéralement aux mêmes obkets en mémoire. Les symboles sont uniques (`'fred` est toujours évalué au même objet). Deux symboles qui se ressemblet sont `eq`. Deux variables référant au même objet sont `eq`.
2. `eqv?` se comporte comme `eq?` mais se comporte correctement lorsqu'il s'agit de comparer des nombres. `eqv?` retourne `#t` si et seulement si les arguments sont `eq` et sont des nombres ayant la même valeur. `eqv?` ne convertit pas les entiers en flottants lors de la comparaison de ces deux types de nombres en revanche.
3. `equal?` retourne `#t` si les arguments ont la même structure. Formellement, on peut définir `equal?` de manière récursive. `equal?` retourne `#t` si et seulement si les arguments sont `eqv`, ou si les arguments sont des listes dont les éléments sont `equal`. Deux objets

qui sont eq sont à la fois eqv et equal. Deux objets qui sont eqv sont equal, mais pas nécessairement eq. Deux objets qui sont equal ne sont pas nécessairement ev ou eq. eq est parfois appelé un comparateur d'identité, et equal un comparateur d'égalité.

Exemples :

```
(define clam '(1 2 3))
(define octopus clam)              ; clam and octopus refer to the same list

(eq? 'clam 'clam)            => #t
(eq? clam clam)              => #t
(eq? clam octopus)           => #t
(eq? clam '(1 2 3))          => #f ; (or (), in MIT Scheme)
(eq? '(1 2 3) '(1 2 3))      => #f
(eq? 10 10)                  => #t ; (generally, but implementation-dependent)
(eq? 10.0 10.0)              => #f ; (generally, but implementation-dependent)
(eqv? 10 10)                 => #t ; always
(eqv? 10.0 10.0)             => #t ; always
(eqv? 10.0 10)               => #f ; no conversion between types
(equal? clam '(1 2 3))       => #t
(equal? '(1 2 3) '(1 2 3))   => #t
```

Scheme fournit = pour la comparaison de deux nombres, et il s'assure de la conversion de types dans ce cas. Par exemple, `(equal? 0 0.0)` renvoit `#f`, alors que ( 0 0.0)= retourne `#t`.

## *Opérateurs logiques*

Scheme fournit plusieurs opérateurs logiques, incluant `and`, `or` et `not`. Les opérateurs `and` et `or` sont des formes spéciales et n'évaluent pas nécessairement tous leurs arguments. Ils évaluent uniquement le nombre d'arguments nécessaires pour décider de retourner `#t` ou `#f` (comme les opérateurs `&&` et `||` en C++). Toutefois il est facile d'écrire une evrsion qui évalue tous les arguments.

```
(and expr1 expr2 ... expr-n)
; return true if all the expr's are true
; ... or more precisely, return expr-n if all the expr's evaluate to
; something other than #f.  Otherwise return #f

(and (equal? 2 3) (equal? 2 2) #t)  => #f

(or expr1 expr2 ... expr-n)
; return true if at least one of the expr's is true
; ... or more precisely, return expr-j if expr-j is the first expr that
```

```
; evaluates to something other than #f.  Otherwise return #f.

(or (equal? 2 3) (equal? 2 2) #t)  => #t

(or (equal? 2 3) 'fred (equal? 3 (/ 1 0)))  => 'fred

(define (single-digit x)
   (and (> x 0) (< x 10)))

(not expr)
; return true if expr is false

(not (= 10 20))     => #t
```

### *Cas particuliers des booléens*

Dans la version R4 de Scheme la liste vide est équivalente à `#f`, et tout le reste est équivalent à `#t`. Par contre, dans R5 la liste vide est également équivalente à `#t`. Moralité : n'utiliser `#f` et `#t` que dans le cas des constantes booléennes.

## *Conditionnelles*

### *Forme spéciale* `if`

```
(if condition true_expression false_expression)
```

Si `condition` est évaluée à `#t` alors le résultat de l'évaluation de `true_expression` est renvoyé ; le cas échéant, le résultat de l'évaluation de `false_expression` est retourné. `if` est une forme spéciale, comme `quote`, car elle n'évalue pas automatiquement tous ses arguments.

```
(if (= 5 (+ 2 3)) 10 20)     => 10
(if (= 0 1) (/ 1 0) (+ 2 3)) => 5
; note that the (/ 1 0) is not evaluated

(define (my-max x y)
   (if (> x y) x y))

(my-max 10 20)               => 20

(define (my-max3 x y z)
   (if (and (> x y) (> x z))
       x
       (if (> y z)
```

```
        y
        z)))
```

### *Une conditionnelle plus générale :* `cond`

La syntaxe de la forme spéciale `cond` est données ci-dessous :

```
(cond (test1 expr1)
      (test2 expr2)
      ....
      (else exprn))
```

Dès que l'on trouve un test qui évalue à `#t`, on évalue l'`expr` correspondante et on retourne sa valeur. Les tests suivants ne sont pas évalués et les autres `expr=s ne sont pas évaluées. Si aucun des tests n'évalue à =#t`, on évalue `exprn` (la partie "else") et on retourne sa valeur. Cette clause est optionnelle mais il est d'usage de la faire apparaître.

```
(define (weather f)
   (cond ((> f 80) 'too-hot)
 ((> f 60) 'nice)
 ((< f 35) 'too-cold)
 (else 'typical-seattle)))
```

### *Style de commentaires*

Si Scheme trouve une ligne de texte avec un point-virgule, tout ce qui suit ce dernier est traité comme un espace. Toutefois, la convention d'usage veut que le point-virgule soit réservé aux commentaires courts portant sur une ligne de code, deux points-virgules sont utilisés pour un commentaire à l'intérieur d'une fonction sur une même ligne, et trois points-virgules sont utilisés dans le cas d'un commentaire introductif ou global (en dehors de la définition d'une fonction).

### *Récursivité et programmation applicative*

On s'intéressera aux aspects suivants : introduction à la récursion, modèles de récursion typiques, trois classes de fonctions récursives sur des listes (*mapping*, *reducing*, *filtering*), fonction d'ordre supérieur et programmation applicative, closures [5] lexicales.

5. Voir note 2

Quelques exemples de fonctions récursives :

```
(define (factorial n)
  (if (= 0 n)
      1
```

```
      (* n (factorial (- n 1)))))

;;  (double-each '(1 3 4))  => (2 6 8)
(define (double-each s)
  (if (null? s)
      ()
      (cons (* 2 (car s))
    (double-each (cdr s)))))
```

*Débogage*

— `trace` permet d'observer les fonctions lors de leurs appels ainsi que leurs valeurs de retour

— `(debug)` appelle le débogeur lorsque l'on reçoit une expression erronnée

Exemple d'utilisation :

```
(trace factorial)
```

*Règles d'écriture des fonctions récursives*

— définir quand s'arrêter (cas de base)

— décider comment progresser vers le cas de base

— formuler la solution en termes d'une étape unique, composée d'une version réduite du problème initial

Pour les nombres, le cas de base est généralement une petite constante numérique, et la evrsion réduite du problème se réduit à `n-1`. Pour les listes, le cas de base est le plus souvent la liste vide, et la evrsion réduite du problème est généralement le reste ou `cdr` de la liste. Voici un modèle de récursion générique :

```
(define (fn args)
 (if base-case
     base-value
   (compute-result (fn (smaller args)))))
```

## *Modèle de récursion*

### *Récursion par accumulation*

De nombreuses fonctions récursives construise leur solution petit à petit. On les appelle *augmenting recursive functions*.

```
;; general form
```

```
(define (func x)
  (if end-test
      end-value
      (augmenting-function augmenting-value (func reduced-x))))
```

La factorielle en constitue un exemple classique :

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

De nombreuses fonctions Scheme opèrent sur des listes, et l'on retrouve fréquemment le cas de la récursion par accumulation où la fonction d'accumulation est `cons` et le cas de base est constitué de la liste vide. On appelle parfois de telles fonctions des fonctions de *mapping*, parce qu'elles appliquent une même fonction à chaque élément de la liste. Notons que puisque l'on utilise `cons`, on construit en réalité une nouvelle liste, qui constituera la valeur de retour. En d'autres termes, on ne modifie pas la liste initiale.

```
(define (double-each s)
  (if (null? s)
      ()
      (cons (* 2 (car s)) (double-each (cdr s)))))
```

Avec la récursion par accumulation de type `cons`, la valeur de base (la valeur retournée lorsque l'on atteint le cas de base) n'est pas touours `()` :

```
;; still cons as augmenting-function, but end-value is not ()

(define (my-append x y)
  (if (null? x) y
      (cons (car x) (my-append (cdr x) y))))
```

Un autre cas de figure important est fourni par les fonctions de réduction, car elles réduisent une liste d'éléments à un élément simple :

```
;; sum the elements of a list

(define (sumlist x)
  (if (null? x) 0
      (+ (car x) (sumlist (cdr x)))))
```

*Récursion terminale*

Dans la récursion terminale (*tail recursion*), on ne construit pas la solution mais on retourne un appel récursif à une version réduite du problème initial. La récursion terminal double-test est la forme la plus courante :

```
;; general form:

(define (func x)
  (cond (end-test-1 end-value-1)
(end-test-2 end-value-2)
(else (func reduced-x))))

;; example: are all elements of a list positive?

(define (all-positive x)
  (cond ((null? x) #t)
((<= (car x) 0) #f)
(else (all-positive (cdr x)))))
;; (all-positive '(3 5 6))  => #t
;; (all-positive '(3 5 -6))  => #f
```

Certaines fonctions récursives terminales disposent de plus d'un argument. Tous les arguments sauf un sont passés sans modification.

```
;; example: member:

(define (my-member e x)
  (cond ((null? x) #f)
((equal? e (car x)) #t)
(else (my-member e (cdr x)))))
```

Une forme moins fréquente est la récursion terminale simple-test.

Les compilateurs Scheme traite la récursion terminale de manière optimisée, comme un programme qui utiliserait des boucles au lieu de la récursion. [6]

6. En particulier, les fonctions récursives terminales n'utilise pas d'espace sur la pile pour chaque appel récursif.

*Récursion simultanée sur plusieurs variables*

```
;; test whether two lists have the same length

(define (same-length x y)
  (cond ((and (null? x) (null? y)) #t)
((null? x) #f)
((null? y) #f)
```

```
(else (same-length (cdr x) (cdr y)))))

;; do these two objects have the same shape?
(define (same-shape x y)
  (cond ((and (pair? x) (pair? y))
 (and (same-shape (car x) (car y))
      (same-shape (cdr x) (cdr y))))
((or (pair? x) (pair? y)) #f)
(else #t)))
```

*Accumulation conditionnelle*

Un dernier modèle important consiste en l'accumulation conditionnelle, où l'on ne procède pas nécessairement à une accumulation à chaque étape. De telles fonctions sont parfois appelées des fonctions de filtrage (*filtering*) car elles suppriment les éléments de la liste qui ne remplissent pas certaines conditions.

```
define (func x)
  (cond (end-test end-value)
(aug-test (augmenting-function augmenting-value (func reduced-x)))
(else (func reduced-x))))

;; example:  remove all non-positive numbers from a list

(define (positive-numbers x)
  (cond ((null? x) ())
((> (car x) 0) (cons (car x) (positive-numbers (cdr x))))
(else (positive-numbers (cdr x)))))
```

*Exemple : tri par insertion*

```
;; variation on list-consing recursion
(define (insert x s)
   (cond ((null? s) (list x))
 ((< x (car s)) (cons x s))
 (else (cons (car s) (insert x (cdr s))))))

;; augmenting recursion
(define (isort s)
   (if (null? s) ()
       (insert (car s) (isort (cdr s)))))
```

### *Programmation applicative*

La programmation applicative repose sur des fonctions de degré supérieur, qui sont des fonctions acceptant d'autres fonctions en arguments ou qui retourne des fonctions en tant que résultat. Scheme tire toute son expressivité et sa capacité d'abstraction en supportant un style de programmation applicative. Scheme considère les fonctions comme des citoyens de première classe.

#### *Appliquer une fonction à une liste*

Considérons l'exemple suivant :

```
;; here is a general function that adds N to every element of a list

(define (addN-to-list alist n)
  (cond ((null? alist) ())
(else (cons (+ n (car alist)) (addN-to-list (cdr alist) n)))))

;; here is a function that takes the cdr of every element of a list
;; (the argument must be a list of lists)
(define (getcdrs alist)
  (cond ((null? alist) ())
(else (cons (cdr (car alist)) (getcdrs (cdr alist))))))
```

Si l'on réfléchit à ce que font ces fonctions à un niveau abstrait, on constate qu'elles appliquent une même fonction à chaque élément d'une liste et qu'elles retournent la liste des valeurs résultantes. Scheme fournit une fonction intégrée supportant ce type d'approche.

`map` est une fonction intégrée au langage qui accepte une fonction et une liste en argument, et retourne la liste résultant de l'application de cette fonction à chaque élément de la liste.

```
(map function list)             ;; general form
(map null? '(3 () () 5))        => (() T T ())
(map round '(3 3.3 4.6 5))      => (3 3 5 5)
(map cdr '((1 2) (3 4) (5 6)))  => ((2) (4) (6))
```

#### *Lambda*

On se retrouve souvent à appliquer de petites fonctions à des listes pour lesquelles il serait fastidieux de définir des fonctions séparées. Par exemple, si l'on souhaite appliquer la fonction `double` aux élements d'une liste, on devrait procéder ainsi :

```
(define (double x) (* 2 x))
```

```
(map double '(3 4 5))         => (6 8 10)
```

La forme spéciale `lambda` peut être utilisée pour définir une fonction anonyme. `lambda` est bel et bien spéciale dans la mesure où elle crée une closure lexicale, qui consiste en un ensemble composé de code et d'un environnement lexical. On verra par la suite en quoi cela est important.

```
(map (lambda (x) (* 2 x)) '(3 4 5))

;; the right way to define addN-to-list
(define (addN-to-list alist n)
  (map (lambda (x) (+ n x)) alist))
```

Notons que dans la fonction `addN-to-list`, le corps de la fonction `lambda` peut adresser la variable `n` qui est dans la portée lexicale de l'expression `lambda`.[7]

7. Ces closures `lambda` sont équivalentes aux blocs en Smalltalk.

`map` peut également être utilisé avec des fonctions qui accepte plus d'un argument. Par exemple :

```
(map + '(1 2 3) '(10 11 12))   => (11 13 15)
(map (lambda (x y) (list x y)) '(a b c) '(j k l))   => ((a j) (b k) (c l))
```

### *Définir des fonctions d'ordre supérieur*

Supposons que l'on souhaite définir une version à un argument de `map`. On pourrait procéder ainsi :

```
(define (my-map func alist)
   (if (null? alist)
       ()
       (cons (func (car alist)) (my-map func (cdr alist)))))
```

### *Closures lexicales*

A présent que l'on dispose de `my-map`, on peut précsier en quoi `lambda` est si spécial. On rappelle qu'une expression `lambda` est évaluée comme une closure lexicale, qui consiste dans le couplage de code et d'un environnement lexical (une portée lexicale, essentiellement). Cet environnement lexical est nécessaire car le code nécessite un endroit pour rechercher la définition des symboles qu'il référence. Par exemple, considérons à nouveau `addN-to-list` ci-dessous :

```
(define (addN-to-list alist n)
  (my-map (lambda (x) (+ n x)) alist))
```

Lorsque l'expression `lambda` est utilisée dans `my-map`, elle nécessite de savoir où chercher la variable nommée `n`. Elle peut obtenir la bonne valeur de `n` car elle retient l'environnement lexical.

### *Environnements emboîtés*

On peut disposer d'autant d'environnements emboîtés que l'on veut. De plus, comme les noms de fonction sont liés de la même manière que celui des variables, les noms de fonction obéissent aux règles de portée.

Par exemple, définissons une simple fonction `test` :

```
(define (test x)
   (+ x 1))
```

L'évaluation de `(test 10)` donne 11.

Cependant, si l'on évalue

```
(let ((test (lambda (x) (* x 2)))
     (test 10))
```

on obtient 20 : à l'intérieur de `let`, `test` est re lié à une fonction différente.

## *Eval*

La fonction `eval` accepte un objet Scheme et un environnement, et elle évalue cet objet. Par exemple :

```
(define fn '*)
(define x 3)
(define y (list '+ x 5))
(define z (list fn 10 y))
x  =>  3
y  => (+ 3 5)
z  => (* 10 (+ 3 5))
(eval '(+ 6 6) user-initial-environment)  => 12
(eval y user-initial-environment)  => 8
(eval z user-initial-environment)  => 80
```

Voici une autre illustration avec des variables dont les valeurs sont atomes :

```
(define a 'b)
(define b 'c)
(define c 50)
a => b
```

```
(eval a user-initial-environment)  => c
(eval (eval a user-initial-environment) user-initial-environment)  => 50
```

Le *top level* de l'interpréteur Scheme est une boucle *read-eval-print* : lecture d'une expression, avluation de cette expression, et affichage du résultat.

`user-initial-environment` est lié à un environnement et est prédéfini. Il existe également des fonctions permettant d'obtenir l'environnement de n'importe quelle procédure. [8]

Les quotes suppriment l'évaluation ; `eval` entraîne l'évaluation. Les deux peuvent s'annuler mutuellement :

```
(define x 3)
x  =>  3
'x => x
(eval 'x user-initial-environment)  => 3
```

## *Apply*

La fonction `apply` permet d'appliquer une fonction à la liste de ses arguments. Par exemple :

```
(apply factorial '(3))  => 6
(apply + '(1 2 3 4))  => 10
```

Une astuce de programmation utile consiste à utiliser `apply` pour définir une fonction qui prend une liste d'arguments lorsque l'on dispose d'une fonction qui accepte un nombre arbitraire d'arguments. Par exemple :

```
(define (sum s) (apply + s))
(sum '(1 2 3))  => 6
```

## *Effets de bord en Scheme*

### *Utilisation des effets de bord*

Les effets de bords sont utilisés :

— pour gérer les entrées-sorties

— lorsque la structure du programme est claire, par exemple mettre à jour d'une petite partie d'une plus large structure représentant l'information concernant l'état actuel du monde

— lorsque l'efficacité est indispensable

8. Voir la section du manuel utilisateur de MIT Scheme sur les boucles read-eval-print.

*Entrées-sorties*

Scheme inclut différentes déclarations pour les entrées et les sorties, incluant les entrées-sorties sur terminal ou sur fichier. Quelques-une des fonctions incluent la lecture, l'écriture et l'affichage de données.

`read` est une fonction qui lit un objet Scheme à partir de l'entrée standard et retourne sa valeur. Il s'agit donc d'une fonction avec un effet de bord.

Exemple :

```
(define clam (read))
```

Il existe d'autres fonctions permettant de lire des caractères sans les décomposer en objets Scheme.

`write` affiche la représentation de son argument sur la sortie standard :

```
(write (+ 3 4))  => 7
```

`display` est comparable à `write` mais n'inclut pas de guillemets autour des chaînes de caractères.

*Formes multiples*

Plusieurs des constructions discutées plus haut, incluant `define`, `let` et `cond`, autorisent l'usage de formes multiples dans leur corps. Toutes ces formes sauf la dernière sont évaluées juste pour leur effet de bord, and la valeur de la dernière forme est utilisée. Par exemple :

```
(define (testit x)
  (write "this function doesn't do much")
  (newline)
  (write "but it does illustrate the point")
  (newline)
  (+ x x))
```

Ici, les expressions `write` et `newline` sont évaluées (pour leur effet) et ensuite, la dernière expression `(+ x x)` est évaluée et sa valeur retournée en tant que valeur de la fonction. De manière similaire, il peut y avoir plusieurs formes dans le corps d'un `let`, ou la clause conséquente d'un `cond`. Toutefois, `if` ne doit avoir qu'une seule forme dans chacun de ses deux parties.

*Assignation et opérations sur les listes*

Scheme possède une forme spéciale pour l'assignation :

```
(set! var expr)
```

Exemples :

```
(define x 10)
(write x)
(set! x (+ x 1))
(write x)
```

Il existe également les fonctions `set-car!` et `set-cdr!` pour mettre à jour le `car` et le `cdr` d'une cellule `cons`. Enfin, il existe une forme spéciale `do` pour les itérations.

Exemple :

```
(define x '(1 2 3 4))
(write x)     ; prints (1 2 3 4)
(set-car! x 100)
(write x)     ; prints (100 2 3 4)
(set-cdr! x ())
(write x)     ; prints  (100)
```

Création d'une liste circulaire :

```
(define circ '(a b))
(set-cdr! (cdr circ) circ)
```

Lorsque l'on passe une liste à une fonction en Scheme, une référence à cette liste est passée, plutôt qu'une copie. Par conséquent, si on utilise `set-car!` ou `set-cdr!` pour mettre à jour une liste à l'intérieur d'une fonction, le paramètre courant sera effectivement modifié.

```
(define (change-it x)
    (set-car! x 'frog))

(define animals '(rabbit horse squirrel monkey))
(change-it animals)
```

Toutefois, considérons l'illustration suivante :

```
(define (test x)
   (write x)
   (set! x 100)
   (write x))

(define y '(1 2 3))
(test y)  ; prints (1 2 3) then 100
(write y) ; prints (1 2 3) ... y is unaltered
```

*Portée et effets de bords*

On peut simuler la programmation orientée objet en Scheme en utilisant une combinaison de la portée lexicale et des effets de bord.

Premièrement, regardons comment une fonction peut partager son état :

```
(define incr)  ; just define the variable incr, but don't give it a value yet
(define get)

(let ((n 0))
   (set! incr (lambda (i) (set! n (+ n i))))
   (set! get (lambda () n)))
```

Les variables `incr` et `get` sont des variables globales, à présent liées à des fonctions. La variable `n` est partagée entre les deux, mais elle reste masquée : il ne s'agit pas d'une variable globale.

A présent, essayons d'évaluer les expressions suivantes :

```
(get)

(incr 10)
(get)

(incr 100)
(get)
```

Il est possible alors d'ajouter un système de distribution (*dispatching*) pour simuler un objet simple. [9] On définit un objet compte bancaire, avec un champ `my-balance`, et les méthodes `balance`, `deposit` et `withdraw` :

9. L'exemple est adapté de [1].

```
(define (make-account)
   (let ((my-balance 0))

      ;; return the current balance
      (define (balance)
 my-balance)

      ;; make a withdrawal
     (define (withdraw amount)
(if (>= my-balance amount)
   (begin (set! my-balance (- my-balance amount))
   my-balance)
   "Insufficient funds"))
```

```
     ;; make a deposit
     (define (deposit amount)
(set! my-balance (+ my-balance amount))
my-balance)

     ;; the dispatching function -- decide what to do with the request
     (define (dispatch m)
(cond ((eq? m 'balance) balance)
      ((eq? m 'withdraw) withdraw)
      ((eq? m 'deposit) deposit)
      (else (error "Unknown request -- MAKE-ACCOUNT"  m))))

      dispatch))
```

Notons que la variable `my-balance` est locale à la fonction `make-account`.

Utilisation du compte bancaire :

```
(define acct1 (make-account))
(define acct2 (make-account))
((acct1 'balance))               => 0
((acct1 'deposit) 100)           => 100
((acct1 'withdraw) 30)           => 70
((acct1 'withdraw) 200)          => "Insufficient funds"

;; acct2 is a different account from acct1!
((acct2 'balance))               => 0
```

---

## *Exercices*

1. Comment les expressions Scheme suivantes sont-elles évaluées ?
   - `(* 2 (+ 4 5))`
   - `( 3 (+ 1 3))=`
   - `(car '(elmer fudd daffy duck))`
   - `(cdr '(elmer fudd daffy duck))`
   - `(and ( 1 2) (= 10 (/ 1 0)))=`
2. Définir une fonction Scheme qui calcule la moyenne de deux nombres.
3. Définir une fonction Scheme `mymax` pour trouver le maximum de deux nombres.
4. Définir une fonction Scheme `sign` pour indiquer le signe d'un nombre (-1 s'il est négatif, 0 s'il vaut 0, 1 s'il est positif)

5. Supposons que nous évaluions ces deux expressions Scheme :

```
(define x '(snail clam))
(define y '(octopus squid scallop))
```

   Dessiner sous forme de diagrammes en boîtes+flèches le résultat de l'évaluation des expressions suivantes. Quelles sont les parties de la liste qui sont créées *de novo*, et quelles sont celles qui sont partagées avec les variables `x` et `y` ?

   — `(cons 'geoduck x)`

   — `(cons y y)`

   — `(append x y)`

   — `(cdr y)`

6. Quel est le résultat de l'évaluation des expressions Scheme suivantes ?

```
(let ((x (+ 2 4))
      (y 100))
   (+ x y))
```

```
(let ((x 100)
      (y 5))
   (let ((x 1))
      (+ x y)))
```

7. Définir une fonction `mylength` qui renvoit la longueur d'une liste.

8. Définir une fonction récursive `add1` qui prend en entrée une liste de nombres et renvoit une nouvelle liste de nombres, chacun incrémenté de 1. Par exemple, `(add1 '(10 20 30))` doit renvoyer `(11 21 31)`.

9. Définir une version non récursive de `add1` qui utilise `map` et `lambda`.

10. A l'aide de `map` et `lambda`, définir une fonction `averages` qui accepte deux listes de nombres et retourne une liste de la moyenne de chaque paire. Par exemple :

```
(averages '(1 2 3) '(11 12 13)) => (6 7 8)
```

11. Quelle est la valeur de `x` et de `y` après avoir évalué les expressions suivantes ?

```
(define x '(1 2 3 4))
(define y x)
(set-car! (cdr x) 100)
```

```
(define x '(1 2 3 4))
(define y x)
(set! x '(100 200))
```

12. Aloysius est intrigué par le code suivant :

```
(define incr)
(define get)
(let ((n 0))
   (set! incr (lambda (i) (set! n (+ n i))))
   (set! get (lambda () n)))
```

Aloysius n'est pas sûr de comprendre comment les fonctions `incr` et `get` fonctionnent, si `n` est sur la pile ou le tas, pourquoi `incr` et `get` continuent de fonctionner correctement même après avoir évalué le `let` ?

Y'a t-il quelque chose de spécial au sujet `let` au niveau du `top level` ? Ainsi, Aloysius expérimente le code suiva,t :

```
(define newincr)
(define newget)
(define (test k)
  (let ((n 0))
     (set! newincr (lambda (i) (set! n (+ n i k))))
     (set! newget (lambda () n))))
```

Ensuite, il évalue `(test 100)`. Cette fois, le `let` est intégré à une fonction et donc Aloysius se dit qu'il est sur la pile mémoire et qu'il disparaîtra lorsque `test` retourne son résultat.

Quel est le résultat attendu lorsque Aloysius évalue chacune des expressions suivantes ?

```
(newget)
(newincr 10)
(newget)
```

Expliquer.

13. Définir une version récursive terminale de `map` pour des fonctions acceptant un seul argument.(Eviter les effets de bord dans la mesure du possible, mais leur usage n'est pas interdit.)

## *Références*

[1] Harold ABELSON et Gerald Jay SUSSMAN. *Structure and Interpretation of Computer Programs*. 2e éd. MIT Press, 1996.

[2] Laurent BLOCH. *Initiation à la programmation avec Scheme*. Technip, 2000.

[3] Jacques CHAZARAIN. *Programmer avec Scheme*. International Thomson Publishing, 1996.