PROGRAMMATION

FONCTIONNELLE

2015

P. Gribomont

1. Introduction et motivation

Premier objectif : Renforcer et appliquer les connaissances acquises

Deuxième objectif : Structuration d'une application en procédures

Troisième objectif : Autonomie en programmation, de la définition du problème à l'utilisation du programme

En bref . . . Pour réussir, il faudra pouvoir programmer et spécifier !

Le moyen

Programmation fonctionnelle en Scheme (Exercices !)

Référence :

P. Gribomont,
Eléments de programmation en Scheme,
Dunod, Paris, 2000.

Version en ligne :

`http://www.montefiore.ulg.ac.be/~gribomon/cours/info0540.html`

Complément très utile :
L. Moreau, C. Queinnec, D. Ribbens et M. Serrano,
Recueil de petits problèmes en Scheme,
Springer, collection Scopos, 1999.

Voir aussi : `http://deptinfo.unice.fr/~roy/biblio.html`

Pourquoi la programmation fonctionnelle ?

— Bon rapport simplicité / généralité
— Plus “naturel” que l’approche impérative
— Favorise la construction et la maintenance

Pourquoi un nouveau langage ?

— L’important est la programmation, pas le langage . . .
donc il faut pouvoir changer de langage
— C comporte des lacunes gênantes
(procédures, structures de données, . . .)

Pourquoi Scheme **(dialecte de** Lisp**) ?**

— Scheme est très simple !
— Scheme est puissant
— Scheme comble des lacunes de C
— Scheme est un langage et un métalangage

Remarque. Tout est relatif ; Scheme est simple, si on considère la puissance du langage ! Bien comprendre le système est difficile, même si on se limite à un sous-ensemble du langage. Obtenir une implémentation efficace en mémoire et en temps est très difficile.

C **et** Scheme **(**Pascal **et** Lisp**)**

Pascal is for building pyramids — imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms — imposing, breathtaking, dynamic structures built by squads fitting fluctuating myriads of simpler organisms into place.

Alan J. PERLIS

Premier lauréat du prix TURING

(Avant-propos de *Structure and Interpretation of Computer Programs*, par Abelson et Sussman, MIT Press & McGraw-Hill, 1985)

2. Les bases de Scheme

Interpréteur : boucle **Read-Eval-Print**

L'interpréteur **lit** une *expression* e, l'**évalue** et **imprime** sa *valeur* $[[e]]$
(ou un message spécifique).
Il peut aussi **lire** une "définition" et l'enregistrer.

Une expression simple est une constante (numérique ou non) ou une variable.
Une expression composée, ou *combinaison*, comporte un opérateur et des arguments ;
les composants sont eux-mêmes des expressions, simples ou composées.
Une combinaison commence par "(" et finit par ")".
Le langage des expressions est inclus dans celui des valeurs
(en arithmétique, c'est le contraire).

```
==> (+ 2 5)
:-) 7
```

(On utilise systématiquement la forme préfixée.)
Cette notation implique que `(+ 2 5)` est une expression, et que sa valeur, calculée par l'interpréteur Scheme, est `7`. On écrit `[[(+ 2 5)]]` = `7`.

```
==> (+ (* 3 -2) (- 5 3))
:-) -4
```

3, -2 et 5 sont des expressions simples (nombres) à partir desquelles on forme les combinaisons `(* 3 -2)` et `(- 5 3)`, puis la combinaison `(+ (* 3 -2) (- 5 3))`.

```
==> (/ 8 5)
:-) 8/5

==> (/ 8.0 5)
:-) 1.6

==> (/ 8 3.0)
:-) 2.666666666666665

==> (/ 2 0)
:-) error - datum out of range 0 within proc /
```

(La valeur de `(/ 2 0)` n'existe pas.)

```
==> 3
:-) 3

==> +
:-) # procedure +
```

[La valeur de + existe et est une procédure (ou fonction).]
[Les valeurs fonctionnelles ne sont pas affichables.]

```
==> pi
:-) 3.14159265358979

==> fact
:-) # procedure fact

==> fict
:-) error - unbound variable fict

==> (2 3)
:-) error - bad procedure 2
```

Remarque. La notation Scheme est plus homogène que les notations mathématiques usuelles, où les foncteurs s'emploient de manière préfixée, infixée ou postfixée, ou encore font l'objet de conventions d'écriture spéciales.

Remarque. Dans la session ci-dessus, on suppose que `pi` et `fact` ont été définis par l'utilisateur (on verra comment plus loin), tandis que `fict` n'a pas été défini. Les nombres et opérations classiques sont naturellement prédéfinis.

Remarque. Les messages peuvent varier d'un évaluateur à l'autre, et être plus ou moins informatifs. Un "message d'erreur" peut ne pas comporter le mot `error`.

Remarque. La structure arborescente des expressions se prête à une définition dans le formalisme BNF. En se restreignant aux expressions arithmétiques, on a

⟨A-expression⟩ : := ⟨nombre⟩ | ⟨variable⟩ | ⟨combinaison⟩

⟨combinaison⟩ : := (⟨A-op⟩ [⟨A-expression⟩]*)

⟨A-op⟩ : := + | - | * | /

On voit qu'une combinaison est un assemblage composé (dans l'ordre), d'une parenthèse ouvrante, d'un opérateur arithmétique, d'un certain nombre (0, 1 ou plus) d'expressions arithmétiques et d'une parenthèse fermante.

`define` = liaison à une valeur

"Définition" du nombre π et de son inverse :

```
==> (define pi 3.14159265358979)
:-) ...
==> (define invpi (/ 1 pi))
:-) ...
==> invpi
:-) 0.31830988618379
```

L'évaluation de l'expression `(define symb` e`)` peut provoquer l'impression du message `symb` (ce n'est pas à proprement parler une valeur, c'est pourquoi nous préférons "..."). L'important est la liaison du symbole "défini" `symb` à la *valeur* de l'expression e ; ce n'est pas la valeur éventuelle de l'expression `(define symb` e`)`.

Si le symbole `pi` reçoit une nouvelle valeur (il est "redéfini"), cela n'influera pas sur `invpi`. En effet, un `define` lie son premier argument (un symbole) à la *valeur* de son second argument, telle qu'elle existe *au moment où l'expression* `define` *est évaluée*.

```
==> (define pi 4.0)
:-) ...

==> invpi
:-) 0.31830988618379
```

Symboles

Les symboles sont des “mots” qui ne sont pas des nombres.

Ils ne peuvent contenir les caractères spéciaux suivants :

`( ) [ ] { } ; , " ’ ‘ # /`

Un symbole est généralement utilisé comme variable ;
sa valeur est alors l’objet qui lui est lié (s’il existe).
Le caractère spécial ’ indique que le symbole n’est pas
utilisé comme variable ; il est alors sa propre valeur.

L’écriture “’`pi`” abrège l’écriture “`(quote pi)`”.
(Le rôle de `quote` sera vu plus loin.)

On a par exemple [[`pi`]] = 3.14... et [[’`pi`]] = `pi`.

Exemples

```
==> pi
:-) 3.14159265358979

==> 'pi
:-) pi

==> (quote pi)
:-) pi

==> pi.+2
:-) error - unbound variable pi.+2

==> 'pi.+2
:-) pi.+2
```

Listes I

Syntaxe. Une liste est une suite d'objets séparés par des blancs, entourée d'une paire de parenthèses (le nombre d'objets est la longueur de la liste). Les listes sont des valeurs ; certaines d'entre elles sont aussi des expressions ; toutes les expressions composées sont des listes.
La seule liste de longueur 0 est la liste vide `()`.

Exemples : `()` , `(* 3)` , `(5 (+ (d 4)) ())` , `(+ 3 5)` , `(define pi 3.14)` , ...

- — *Constructeur* **`cons`** (deux arguments) :
  Si [[β]] est une liste, alors [[(`cons` α β)]] est une liste dont le premier élément est [[α]] et le reste est [[β]].
- — *Constructeur* **`list`** (nombre quelconque d'arguments) :
  [[(`list` α_1 ... α_n)]] est la liste dont les éléments sont (dans l'ordre) [[α_1]], ..., [[α_n]].

Les fonctions [[`cons`]] et [[`list`]] sont injectives.

(`list` α_1 α_2 ...α_n) équivaut à
(`cons` α_1 (`cons` α_2 ... (`cons` α_n '()) ...))

Listes II

Notation. Une liste non vide dont le premier élément est x et dont le reste est la liste ℓ peut s'écrire (`x . ℓ`). Les écritures

```
(0 1 2)
(0 . (1 2))
(0 1 . (2))
(0 1 2 . ())
(0 . (1 . (2)))
(0 . (1 2 . ()))
(0 1 . (2 . ()))
(0 . (1 . (2 . ())))
```

représentent toutes la même liste de longueur 3, dont les éléments sont `0`, `1` et `2`.

Cette notation sera généralisée plus loin. On note déjà que

[[`(cons` α β`)`]] = (`[[`α`]] . [[`β`]]`)
[[`(cons 3 7)`]] = `(3 . 7)` (paire pointée)

Reconnaisseurs de listes.

- Reconnaître une liste : prédicat `list?`
  [[`(list? l)`]] est vrai si [[`l`]] est une liste.
- Reconnaître la liste vide : prédicat `null?`
  [[`(null? l)`]] est vrai si [[`l`]] est `()`.

Listes III

Décomposition des listes.

Si [[l]] est une liste non vide, son premier élément est [[`(car l)`]] ;
les éléments restants forment la liste [[`(cdr l)`]].

On a donc [[l]] = [[`(cons (car l) (cdr l))`]].

Si [[a]] est un objet et [[l]] une liste, on a [[`(cons a l)`]] = `([[a]] . [[l]])` et
[[`(car (cons a l))`]] = [[a]] et [[`(cdr (cons a l))`]] = [[l]].

```
(define l (list 3 '5 '(6 7) '3a 'define pi))        ...
l                                        (3 5 (6 7) 3a define 3.14159265358979)
(car l)                                  3
(cdr l)                                  (5 (6 7) 3a define 3.14159265358979)
(cdaddr l)                               (7)
(cons '(a b) '(6 7))                     ((a b) 6 7)
(cons (car l) (cdr l))                   (3 5 (6 7) 3a define 3.14159265358979)
```

Booléens et prédicats I

Les booléens sont des symboles spéciaux :

`#t` pour *true* (vrai) et `#f` pour *false* (faux).

Remarque. Certains systèmes assimilent "`#f`" et "`()`" (liste vide) ;
dans la suite, nous évitons cette assimilation fautive.

Les prédicats sont des fonctions prenant leurs valeurs dans les booléens. Beaucoup de prédicats numériques (`<`, `=`, `zero?`, ...) ou non (`null?`, `eq?`, `equal?`, ...) sont prédéfinis.

Les reconnaisseurs sont des prédicats à un argument, associés aux différentes catégories d'objets.

Voici des reconnaisseurs prédéfinis importants :

```
number?  symbol?  boolean?  list?  procedure?
```

Booléens et prédicats II

Exemples.

```
==> (> 7 3 -2 -5/2)
:-) #t
==> (> 7 -2 3 -5/2)
:-) #f

==> (= (- 3 2) (/ 6 6) 1 (+ 0 1))
:-) #t
==> (equal? l (cons (car l) (cdr l)))
:-) #t
```

Conditionnels

On suppose que, dans l'*environnement* où les expressions suivantes sont évaluées, `pi` est défini (c'est-à-dire, lié à une valeur) et que `pu` ne l'est pas.

```
(if (> 5 3) pi pu)                  3.14159265358979
(if (> 3 5) pi pu)                  Error - unbound variable pu
(if (> pi pu) 3 4)                  Error - unbound variable pu
(cond ((> 0 1) (/ 2 0))
      ((> 1 2) (/ 0 1))
      ((> 2 0) (/ 1 2)))            1/2
(cond)                              Error - no value
(cond ((> 0 1) 5))                  Error - no value
(cond ((> pi 5) trucmuche)
       (else 4))                    4
(cond ('trucmuche 3)
      (else 4))                     3
```

3. Règles d'évaluation

La règle d'évaluation des expressions est récursive :
l'évaluation d'une expression peut requérir
l'évaluation de sous-expressions.

Le processus d'évaluation renvoie en général une valeur ;
il peut aussi avoir un effet (outre l'affichage de la valeur).

Les nombres et les variables sont des cas de base.
Les combinaisons sont des cas inductifs.
Les formes spéciales sont des cas de base ou des cas inductifs.

Cas de base

L'évaluation d'un nombre donne ce nombre.
L'évaluation d'un booléen donne ce booléen.
L'évaluation d'un symbole donne la valeur liée à ce symbole,
si cette valeur existe et est affichable.

```
5/4     ==>     5/4
#f      ==>     #f
pi      ==>     3.14159265358979
pa      ==>     Error - unbound variable
+       ==>     # procedure ...
car     ==>     # procedure ...
fact    ==>     # procedure ...
fict    ==>     Error - unbound variable
```

Cas inductif

L'expression à évaluer est une liste (e_0 ... e_n).

La combinaison

1. Les e_i sont évalués (soient v_i les valeurs) ;
   v_0 doit être une fonction dont le domaine contient $(v_1, \ldots, v_n)$.
2. La fonction v_0 est appliquée à $(v_1, \ldots, v_n)$.
3. Le résultat est affiché.

Une fonction (mathématique) est un ensemble de couples de valeurs ; la première est un élément du domaine, la seconde l'image correspondante.

Appliquer une fonction est produire l'image associée à un élément donné du domaine de la fonction.

Pour les fonctions prédéfinies (liées à `+`, `cons`, etc. dans l'environnement global) le processus d'application est l'exécution du code correspondant à la fonction. Le cas des fonctions définies par l'utilisateur est envisagé plus loin.

Les formes spéciales

La syntaxe est celle d'une combinaison, mais chaque forme spéciale a sa propre règle d'évaluation. Les formes spéciales sont identifiées par un mot-clef, qui est le premier élément de l'expression à évaluer.

Liste partielle

`define`	`let`
`if`	`let*`
`cond`	`letrec`
`quote`	`...`
`lambda`	`...`
`and`	`...`
`or`	`...`

Remarques. Les valeurs des sous-expressions ne sont pas affichées.
En cas d'erreur, l'évaluation est interrompue et un message est affiché.

Exemples de combinaison.

Les sous-expressions de ces combinaisons sont des nombres, des booléens, des symboles, des formes spéciales ou des combinaisons.

Les composants de la combinaison sont évalués (dans un ordre quelconque ou même en parallèle), puis la valeur du premier composant est appliquée aux valeurs des composants suivants.

```
(+ 2 4)                                          6

(+ (- 5 3) (* 2 2))                              6

(cons (car '(5 6)) (cdr (list 7 8 9)))           (5 8 9)

(car (cons (car '(5 6)) (cdr (list 7 8))))       5

(car '(5 8 9))                                   5

(+ (* 4 (car 5)) 3)                              Error - ...
```

Evaluation de `(cond (`c_1 e_1`) ...(`c_n e_n`))`

Les prédicats $c_1, \ldots, c_n$ sont évalués en séquence, jusqu'au premier c_k dont la valeur n'est pas `#f` (ce sera le plus souvent `#t`). La valeur du conditionnel est alors celle de l'expression correspondante e_k.

Si tous les prédicats sont évalués à `#f`, le conditionnel n'a pas de valeur. (C'est généralement lié à une erreur de programmation, même si l'évaluateur n'affiche pas `error`.) Le dernier prédicat c_n peut être le mot-clef `else` dont la valeur, dans ce contexte, est `#t`.

`(if` e_0 e_1 e_2`)` abrège `(cond (`e_0 e_1`) (else` e_2`))`

```
(cond)                          Error - no value
(cond (1 2))                    2
(if (< 5 3) (/ 2 0) 5)          5
(cond ((> 0 1) (/ 2 0))
      ((> 2 0) (/ 1 2)))        1/2
(cond ((> 0 1) 5))              Error - no value
(cond (else 4))                 4
```

Evaluation de `(and` e_1 `...`e_n`)`

Les e_i sont évalués dans l'ordre et la première valeur fausse est retournée ; les valeurs suivantes ne sont pas calculées. S'il n'y a pas de valeurs fausses, la dernière valeur calculée est retournée. S'il n'y a pas de valeurs du tout, `#t` est retourné.

```
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))        #f
(and (= 2 2) (+ 2 3) (/ 2 0) (< 2 1))        Error - Division by zero
(and (= 2 2) (< 2 1) (/ 2 0) (+ 2 3))        #f
(and (= 2 2) (+ 2 3))                        5
(and (+ 2 3) (= 2 2))                        #t
(and)                                        #t
(if (and (= 2 2) (+ 2 3)) 'vrai 'faux)       vrai
```

Evaluation de `(or` e_1 `...`e_n`)`

Les e_i sont évalués dans l'ordre et la première valeur non fausse est retournée ; les valeurs suivantes ne sont pas calculées. S'il n'y a pas de valeurs non fausses, ou pas de valeurs du tout, `#f` est retourné.

```
(or (< 2 1) (= 2 2) (/ 2 0) (+ 2 3))          #t
(or (< 2 1) (+ 2 3) (/ 2 0) (= 2 2))          5
(or (< 2 1) (/ 2 0) (+ 2 3) (= 2 2))          Error - Division by zero
(or (= 2 2) (+ 2 3) (< 2 1))                  #t
(or)                                          #f
(or #f (< 2 1) #f)                            #f
(if (or (+ 2 3) 1 2) 'vrai 'faux)             vrai
```

Evaluation de `(quote` e`)`.
La valeur est simplement e, à ne pas confondre avec la *valeur* de e !
La forme spéciale `(quote` e`)` s'abrège en `'`e.

```
(quote +)                    +
'+                           +
+                            # procedure
(quote 3)                    3
'3                           3
3                            3
(quote pi)                   pi
'pi                          pi
pi                           3.14159265358979
'''pi                        ''pi
(car (a b c))                Error - unbound var
(car '(a b c))               a
(cons 'a '(b c))             (a b c)
(car (cons 'a '(b c)))       a
(car '(cons a (b c)))        cons
(car (cons a (b c)))         Error - unbound var
```

"Evaluation" de la forme **define**.
La valeur de (`define` s e) n'est pas spécifiée.
L'évaluation a pour *effet* de lier la valeur [[e]] au symbole s.
Une forme spéciale `define` peut apparaître à l'intérieur d'une autre forme, mais nous n'utilisons pas cette possibilité par la suite ; les formes spéciales `define` sont normalement évaluées dans l'environnement global et toute nouvelle liaison (par `define`) d'une valeur à un symbole `x` rend caduque la liaison antérieure éventuelle à ce symbole.

```
x                          Error - unbound var
(define x (+ 4 1))         ...
x                          5
(+ x 3)                    8
(define x +)               ...
x                          # procedure
'x                         x
(x 2 3 4)                  9
(define 'x 7)              Error
(define x (x 4 4))         ...
x                          8
```

4. La forme `lambda`

La fonction qui à son unique argument associe le carré de sa valeur s'écrit parfois $\lambda x \ . \ x * x$. (Voir `http://en.wikipedia.org/wiki/Lambda_calculus`.)

La variable x n'a qu'un rôle de marque-place et peut être renommée.

En Scheme on écrit `(lambda (x) (* x x))`.
La liste `(x)` est la *liste des paramètres*.
La liste `(* x x)` est le *corps* de la forme `lambda`.

```
pi                                  3.14159265358979
(lambda (y) (* y y))                # procedure
((lambda (y) (* y y)) 12)           144
((lambda (pi) (* pi pi)) 12)        144
((lambda (car) (* car car)) 12)     144
y                                   Error - unbound variable y
pi                                  3.14159265358979
car                                 # procedure
```

Règle de calcul

Evaluation de `(`e_0 e_1 `...` e_n`)`, e_0 = `(lambda (x1 ... xn)` M`)`.

On évalue d'abord les expressions e_i, ce qui donne les valeurs v_i
($i = 0, 1, \ldots, n$).
La valeur v_0 est une fonction à n arguments.
On applique v_0 à la suite des valeurs $(v_1, \ldots, v_n)$;
le résultat est la valeur de la combinaison.

Le processus d'application est, par définition, le processus d'évaluation de l'expression M (le *corps* de la forme `lambda`), dans laquelle les variables `x1`, ..., `xn` sont liées aux valeurs $v_1, \ldots, v_n$, respectivement.

Exemple.

```
x                                  Error - unbound variable x
y                                  9
((lambda (x y) (+ 2 x y)) 3 7)     12
(+ 2 3 7)                          12
x                                  Error - unbound variable x
y                                  9
```

Remarque importante. La liaison des `xi` aux v_i disparaît dès que le processus d'application se termine. Les liaisons antérieures éventuelles des `xi` sont alors à nouveau en vigueur. Le caractère temporaire des liaisons des paramètres permet d'éviter les "téléscopages" de noms.

Exemple. Dans un environnement où `x` a une valeur numérique, l'expression

```
((lambda (x) (* 2 x))
 ((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x))
```

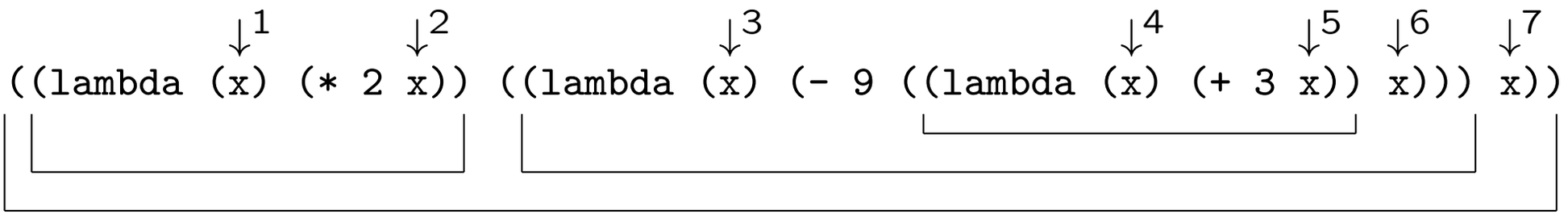
a la même valeur que l'expression

```
((lambda (w) (* 2 w))
 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))
```

(Le fait que `u`, `v` et `w` aient des valeurs ou non dans cet environnement est sans importance.)

Remarque. Un environnement est un ensemble de liaisons variable-valeur.

Portée, réduction et combinaison I



Portée, réduction et combinaison II

```
==> (define x 5)
:-) ...
```

Les expressions qui suivent ont toutes pour valeur 2.

```
((lambda (x) (* 2 x)) ((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x))

((lambda (w) (* 2 w)) ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))

(* 2 ((lambda (v) (- 9 ((lambda (u) (+ 3 u)) v))) x))
((lambda (w) (* 2 w)) (- 9 ((lambda (u) (+ 3 u)) x)))
((lambda (w) (* 2 w)) ((lambda (v) (- 9 (+ 3 v))) x))

(* 2 (- 9 ((lambda (u) (+ 3 u)) x)))
(* 2 ((lambda (v) (- 9 (+ 3 v))) x))
((lambda (w) (* 2 w)) (- 9 (+ 3 x)))

(* 2 (- 9 (+ 3 x)))
```

Occurrences libres, occurrences liées I

Expression mathématique :

$$E =_{def} 1 - \int_0^x \sin(u)\,\mathrm{d}u$$

le symbole x a une valeur (dont dépend celle de E) ; le symbole u n'a pas de valeur ; Remplacer (partout !) u par t ne change pas la valeur de E. Les symboles 1, 0, sin et $\int \ldots \mathrm{d} \ldots$ ont des valeurs (nombres pour les deux premiers, fonction pour le troisième, opérateur d'intégration pour le dernier).

En logique, la valeur de vérité de $\forall x\, P(x, y)$ dépend de la valeur de la "variable libre" y (et de l'interprétation de la constante P) mais pas de la valeur de la "variable liée" x qui peut être renommée : $\forall x\, P(x, y)$ et $\forall z\, P(z, y)$ sont des formules logiquement équivalentes.

Le même genre de distinction doit se faire en Scheme. Pour évaluer l'expression `((lambda (x) (+ x y)) 5)`, il sera nécessaire que `y` ait une valeur ; par contre, la valeur éventuelle de `x` est sans importance. On peut d'ailleurs remplacer les deux occurrences de `x` par `z`, par exemple.

Occurrences libres, occurrences liées II

Les notations Scheme sont sur ce point plus flexibles que les notations mathématiques. On ne peut écrire

$$\int_0^x \sin(x)\,\mathrm{d}x \quad \text{au lieu de} \quad \int_0^x \sin(u)\,\mathrm{d}u$$

mais on peut écrire

`((lambda (x) (+ x 5)) x)` au lieu de `((lambda (u) (+ u 5)) x)`

(même si la seconde notation est naturellement préférable).
L'utilisateur doit se méfier des "téléscopages de noms".

On observe aussi que la valeur de

```
((lambda (u) (+ u 5)) (* x 3))
```

dépend de celle de `x` mais que la valeur de

```
(lambda (x) ((lambda (u) (+ u 5)) (* x 3)))
```

n'en dépend pas.

Occurrences libres, occurrences liées III

Toute occurrence de `x` dans le corps `E` de la forme `(lambda (... x ...) E)` est dite *liée*. Par extension, une occurrence de `x` est liée dans une expression α si cette expression comporte une sous-expression (forme `lambda`) dans laquelle l'occurrence en question est liée. Une occurrence non liée (en dehors d'une liste de paramètres) est *libre*.

Dans l'expression `((lambda (x) (- 9 ((lambda (x) (+ 3 x)) x))) x)`, les deux premières occurrences de `x` sont des paramètres, les troisième et quatrième occurrences de `x` sont liées, la cinquième occurrence de `x` est libre.

Dans l'expression `(* 9 ((lambda (x) (+ 3 x)) x) x)`, la première occurrence de `x` est un paramètre, la deuxième occurrence de `x` est liée, les troisième et quatrième occurrences de `x` sont libres.

Les variables liées sont habituelles en logique, et aussi en mathématique (indice de sommation, variable d'intégration).

Statut “première classe” des procédures

Les procédures, valeurs des formes `lambda`, héritent de toutes les propriétés essentielles des valeurs usuelles comme les nombres. En particulier, on peut définir une procédure admettant d'autres procédures comme arguments, et renvoyant une procédure comme résultat. De même, on peut lier une procédure à un symbole, en utilisant `define`.

Ceci évoque les mathématiques (les domaines de fonctions ont le même “statut” que les domaines de nombres), mais contraste avec les langages de programmation usuels, qui ne permettent généralement pas, par exemple, de renvoyer une procédure comme résultat de l'application d'une autre procédure à ses arguments.

```
(define square (lambda (x) (* x x))) ==>  ...
square                               ==>  # procedure
(square (+ 4 1))                     ==>  25
```

Evaluer `square` et `(+ 4 1)`, ce qui donne les valeurs v_0 et v_1.
Appliquer la valeur v_0 (fonction unaire) à la valeur $v_1 =$ `5`
i.e. évaluer `(* x x)` où `x` est lié à (remplacé par) `5`
i.e. évaluer `(* 5 5)`.

Remarque. Conceptuellement, la valeur-procédure v_0 associée à la variable `square` pourrait être la table (infinie) des couples (n, n^2). Un tel objet n'est pas affichable, et Scheme affichera simplement “`# procedure`”.

Arguments et valeurs fonctionnels I

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))          ...

(compose car cdr)                     # procedure

((compose car cdr) '(1 2 3 4))        2

(cadr '(1 2 3 4))                     2

((compose (compose car cdr) cdr)
 '(1 2 3 4))                          3

((compose (compose car cdr)
           (compose cdr cdr))
 '(1 2 3 4))                          4
```

Arguments et valeurs fonctionnels II

Etant donné une fonction `f` et un incrément `dx`, une approximation de la fonction dérivée en `x` est (la valeur de) `(/ (- (f (+ x dx)) (f x)) dx)`. On peut définir la fonction `deriv-p` (dérivée en un point) comme suit :

```
(define deriv-p
  (lambda (f x dx) (/ (- (f (+ x dx)) (f x)) dx)))
```

On a

```
(deriv-p square 10 0.00001)  ==>  20.0000099
```

Il est plus élégant de définir d'emblée l'*opérateur* `deriv` :

```
(define deriv
  (lambda (f dx)
    (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
```

On a alors

```
((deriv square 0.00001) 10)  ==>  20.0000099
```

5. Récursivité

La notion de fonction est cruciale en programmation parce qu'elle permet de résoudre élégamment de gros problèmes en les ramenant à des problèmes plus simples.

De même, la notion de fonction est cruciale en mathématique parce qu'elle permet de construire et de nommer des objets complexes en combinant des objets plus simples.

La définition $hypo =_{def} \lambda x, y : \sqrt{x^2 + y^2}$
de même que le programme

```
(define hypo
  (lambda (x y) (sqrt (+ (square x) (square y)))))
```

sont des exemples classiques de fonctions. La définition (opérationnelle) de la longueur de l'hypoténuse suppose les définitions préalables de l'addition, de l'élévation au carré et de l'extraction de la racine carrée.

Remarque. L'ordre dans lequel on *définit* les procédures n'a pas d'importance. On peut définir (mais non utiliser) `hypo` avant de définir `square`.

Idée de base I

L'idée de la récursivité est d'utiliser, dans la définition d'un objet relativement complexe, non seulement des objets antérieurement définis, mais aussi l'objet à définir lui-même.
Le risque de "cercle vicieux" existe, mais n'est pas inévitable. On peut faire une analogie avec les égalités et les équations. Dans un contexte où f, a et b sont définis, on peut définir x par l'égalité

$$x = f(a, b)$$

Par contre, l'égalité

$$x = f(a, x)$$

ne définit pas nécessairement un et un seul objet x ; même si c'est le cas, le procédé de calcul de x peut ne pas exister.
Un procédé de calcul parfois utilisé pour résoudre l'équation $x = f(a, x)$ consiste à construire un préfixe plus ou moins long de la suite

$$x_0,\ x_1 = f(a, x_0),\ x_2 = f(a, x_1),\ \ldots,\ x_{n+1} = f(a, x_n), \ldots$$

et à considérer que le dernier terme calculé est (proche de) la solution cherchée. Ce procédé fournit rapidement, par exemple, une bonne approximation de la solution de l'équation $x = \cos x$.

Idée de base II

Le procédé d'approximation peut aussi être utilisé pour calculer des fonctions plutôt que des nombres, et donc pour résoudre des équations fonctionnelles du type

$$f \;=\; \Phi(g, f)\,.$$

L'équation différentielle $y' = f(x, y)$ (avec $y(x_0) = y_0$) peut s'écrire

$$y(x) \;=\; y_0 + \int_{x_0}^{x} f(t, y(t))\,\mathrm{d}t$$

ce qui permet souvent une résolution approchée.

Exercice. Résoudre l'équation $y' = y$, avec $y(0) = 1$.

Idée de base III

En programmation, on doit à toute construction associer un procédé de calcul clair, précis et raisonnablement efficace. Pour cette raison, la définition récursive de fonctions sera limitée à des instances particulières du schéma $f = \Phi(g, f)$.

En pratique, on constatera que l'évaluation de $f(x)$ nécessitera la détermination préalable d'un ensemble de valeurs $f(y_1), \ldots, f(y_n)$ où les $y_1, \ldots, y_n$ sont, en un certain sens, "plus simples" que x.

Si le domaine de la fonction f à définir est tel que tout élément n'admet qu'un ensemble fini d'éléments "plus simples", on conçoit que le risque de "tourner en rond" pourra être évité.

Deux exemples classiques I

Définition de la factorielle (fonction de $\mathbb{N}$ dans $\mathbb{N}$) :

$$n! \;=\; [\,\text{if } n = 0 \text{ then } 1 \text{ else } n * (n-1)!\,]\,.$$

Exemple d'exploitation de la définition :

$$3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6\,.$$

Deux exemples classiques II

Définition de la suite de Fibonacci :

$$f_n \;=\; [\,\text{if } n < 2 \text{ then } n \text{ else } f_{n-1} + f_{n-2}\,]\,.$$

$$\begin{aligned}
&f_0 = 0\,,\\
&f_1 = 1\,,\\
&f_2 = 1 + 0 = 1\,,\\
&f_3 = 1 + 1 = 2\,,\\
&f_4 = 2 + 1 = 3\,,\\
&f_5 = 3 + 2 = 5\,,\\
&f_6 = 5 + 3 = 8\,,\\
&\ldots
\end{aligned}$$

Dans les deux cas, “plus simple” s’identifie à “plus petit”.
Chaque entier *naturel* n’admettant qu’un nombre fini de naturels plus petits, on conçoit que ces définitions *récursives* de fonctions soient “acceptables” (le calcul se termine).

Deux exemples classiques III

Tout ceci se traduit aisément en `Scheme` :

```
(define fact
  (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))     ...

(fact 0)                                                    1
(fact 4)                                                   24
(fact (fact 4))                          620448401733239439360000
(fact -1)                                          Aborting! ...

(define fib
  (lambda (n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))      ...

(fib  0)                                                    0
(fib  6)                                                    8
(fib 12)                                                  144
```

Double rôle de `define`

La forme **`(define var`** exp**`)`** peut être évaluée dans deux cadres distincts.

- L'expression exp ne comporte pas d'occurrence libre de `var`. Dans ce cas non récursif, exp a une valeur v au moment où la forme `define` est évaluée, et le seul rôle de cette évaluation est de provoquer la liaison de v à `var`.
- L'expression exp est une forme `lambda`, pouvant comporter des occurrences libres de `var`. Dans ce cas, exp n'a pas de valeur (du moins, pas de valeur utile) avant l'évaluation de la forme `define`. Cette évaluation a alors le double rôle de définir une procédure récursive et de la lier à (de lui donner le nom) `var`.

Avant l'évaluation de la forme spéciale

```
(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
```

la forme `(lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))` n'a pas de valeur utile puisque `fact` n'a pas de valeur. Après l'évaluation, la procédure de calcul de la factorielle est liée à (est la valeur de) la variable `fact`.

Processus de calcul I

Simulation d'un calcul récursif :

```
(fact 3)
(if (zero? 3) 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
...
(* 3 (* 2 (fact 1)))
...
(* 3 (* 2 (* 1 (fact 0))))
...
(* 3 (* 2 (* 1 1)))
...
6
```

On conçoit que ce type de calcul requiert un espace-mémoire dont la taille dépend (ici, linéairement) de la valeur de l'argument.
On note aussi que l'introduction de la récursivité ne semble pas exiger de mécanisme particulier pour appliquer une fonction à des arguments : les règles déjà introduites suffisent. Cet avantage peut entraîner des déboires pour l'utilisateur peu soucieux du processus de calcul.

Processus de calcul II

Scheme accepte les définitions

```
(define f (lambda (x) (f x)))
(define g (lambda (x) (g (+ x 1))))
(define h (lambda (x) (+ (h x) 1)))
```

Evaluer `(f 0)`, `(g 1)` ou `(h 2)` ne donne rien . . .
. . . sauf un gaspillage de ressources !
Même chose pour `(fact -1)` : procédure utile, mais argument inapproprié.

L'utilisateur peut éviter ce risque en vérifiant que tout appel pour des valeurs données donne lieu à un nombre fini d'appels subséquents, pour des valeurs "plus simples". Des schémas de programmes récursifs existent, qui garantissent la propriété de terminaison.

Exercice. Que penser, dans le domaine des réels positifs, de la définition suivante :

$$f(0) = a\,, \quad f(x) = g(b, f(x/2)) \text{ si } x > 0\,?$$

Processus de calcul III

La terminaison du calcul de `(fib` n`)` ($n \in \mathbb{N}$) : se démontre par récurrence.

L'efficacité du calcul est inacceptable :

```
(fib 9)
(+ (fib 8) (fib 7))
(+ (+ (fib 7) (fib 6)) (fib 7))
...
```

L'expression `(fib 7)` sera évaluée deux fois. En fait,
1 évaluation de `(fib 9)` implique
1 évaluation de `(fib 8)`,
2 évaluations de `(fib 7)`,
3 évaluations de `(fib 6)`,
5 évaluations de `(fib 5)`, etc.

Temps de calcul : proportionnel à la valeur calculée $f_n \simeq 1.6^n$.

Deux remèdes possibles seront vus plus loin :
1) Utiliser un autre principe de calcul (moins naïf) ;
2) Forcer l'interpréteur à réutiliser les résultats intermédiaires.

Processus de calcul IV

On sait que les nombres de Fibonacci peuvent se calculer plus efficacement, au moyen d'une simple boucle. L'algorithme correspondant se traduit immédiatement en scheme :

```
(define fib-a
  (lambda (n a b)
    (if (zero? n)
        a
        (fib-a (- n 1) b (+ a b)))))
```

```
(define fib
  (lambda (n)
    (fib-a n 0 1)))
```

Si `n`, `a` et `b` ont pour valeurs respectives n, f_i et f_{i+1}, alors `(fib-a n a b)` a pour valeur f_{n+i}.
En particulier, `(fib-a n 0 1)` a pour valeur f_n.

Signalons déjà que ce type de récursivité "dégénérée" se reconnaît par une simple analyse de la syntaxe du programme. Cette analyse permet à l'interpréteur d'exécuter le programme aussi efficacement qu'une simple boucle en C, sans consommation inutile de mémoire.

Récursivité croisée

Exemple classique.

```
(define even?
  (lambda (n) (if (zero? n) #t (odd? (- n 1)))))

(define odd?
  (lambda (n) (if (zero? n) #f (even? (- n 1)))))
```

Processus de calcul.

```
(odd? 4) -> (even? 3) -> (odd? 2) -> (even? 1) -> (odd? 0) -> #f
```

Rappel. L'ordre dans lequel on définit les procédures n'a pas d'importance. Seule exigence (naturelle) : les procédures doivent être définies avant d'être utilisées (appliquées à des arguments). Le double rôle de `define` existe dès qu'il y a récursivité, croisée ou non.

6. Récursivité structurelle

Principe de la récursivité structurelle

Le système "accepte" toute définition récursive syntaxiquement correcte, même si la procédure associée donne lieu à des calculs infinis. L'utilisateur doit savoir si, dans un domaine donné, le calcul se terminera toujours. Pour les domaines usuels, des schémas existent qui garantissent la terminaison.

Cas particulier : les schémas *structurels*, basés sur la manière dont les objets du domaine de calcul sont construits.

Le processus d'évaluation réduit le calcul de $f(v)$ au calcul de $f(v_1), \ldots, f(v_n)$ où les v_i sont des *composants (immédiats)* de v. Cette technique est sûre tant que l'on se limite aux domaines dont les objets ont un nombre fini de composants (immédiats ou non), clairement identifiés.

Domaines usuels de base : Nombres naturels
Listes
Expressions symboliques

En plus : domaines dérivés des précédents, surtout par produit cartésien.

Récursivité structurelle : les naturels I

Conceptuellement, les naturels sont construits à partir de 0 et de la fonction successeur. 0 n'a pas de composant (objet de base) ; 16 est le seul composant immédiat de $17 = succ(16)$.

Schéma de base

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u))
           n
           u))))
```

Les fonctions `G`, `H` et `K` sont supposées déjà définies.
`u` représente une suite de 0, 1 ou plusieurs arguments ;
(une suite de 0 argument se réduit à rien !).

Le calcul de `(F 0 u)` n'implique pas d'appel récursif.
Le calcul de `(F n u)` implique celui de `(F (- n 1) (K n u))` si `n` n'est pas nul.

Récursivité structurelle : les naturels II

Schéma de base simplifié

Le cas où `u` est absent suffit souvent :

```
(define F
  (lambda (n)
    (if (zero? n)
        c
        (H (F (- n 1))
           n))))
```

Récursivité structurelle : les naturels III

Exemples :

```
(define harmonic-sum
  (lambda (n)
    (if (zero? n)
        0
        (+ (/ 1 n) (harmonic-sum (- n 1))))))

(define mult
  (lambda (n u)
    (if (zero? n)
        0
        (+ u (mult (- n 1) u)))))
```

Récursivité structurelle : les naturels IV

Exemples :

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))

(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

Récursivité structurelle : les naturels V

Les schémas sont d'abord des schémas de pensée ; ils suggèrent d'exprimer $f(n)$ en termes d'expressions indépendantes de f, mais aussi de $f(n-1)$, si $n > 0$. Les schémas imposent en outre la syntaxe du programme : le programmeur doit seulement définir les fonctions `G`, `H` et `K`.

Exemple

```
(define cbin
  (lambda (n u)
    (if (zero? n)
        1
        (/ (* (cbin (- n 1) (- u 1)) u) n))))
```

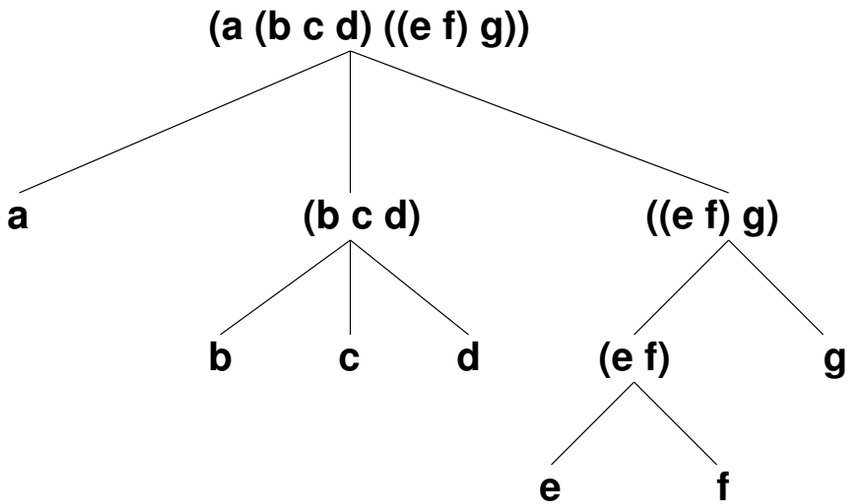
Récursivité structurelle : les naturels VI

Le même exemple . . .

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u)) n u))))

(define G (lambda (u) 1))
(define K (lambda (n u) (- u 1)))
(define H (lambda (r n u) (/ (* r u) n)))
(define cbin F)
```

Listes : représentation arborescente

Il existe une correspondance naturelle entre les listes et les arbres.
Chaque nœud de l'arbre a un nombre fini quelconque de fils.



ATTENTION !
Cette représentation commode des listes n'est pas celle employée en machine !

Remarque. Un arbre réduit à sa racine et étiqueté par un symbole atomique ne sera pas représenté par une liste mais par ce symbole.

Remarque. Conceptuellement, seuls les nœuds terminaux de l'arbre représenté ici sont étiquetés ; les étiquettes attachées aux nœuds internes sont synthétisées à partir des étiquettes des nœuds successeurs. Les arbres dont les nœuds internes sont étiquetés (indépendamment des feuilles) forment un autre type de données.

Listes : récursivité superficielle I

[[l]] est soit `()`, soit [[`(cons (car l) (cdr l))`]].

Toute liste s'obtient à partir de `()` et de `cons` ; la règle de construction est simple : `([[a1]] [[a2]] ... [[an]])` s'obtient en évaluant
`(cons a1 (cons a2 (cons ... (cons an '()) ...)))`

Schéma de base

```
(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u))
           l
           u))))
```

Les fonctions `G`, `H` et `K` sont supposées déjà définies ;
`u` représente une suite de 0, 1 ou plusieurs arguments.

Calculer `(F () u)` n'implique pas d'appel récursif.
Calculer `(F l u)` implique l'appel `(F (cdr l) (K l u))` si `l` n'est pas vide.

Listes : récursivité superficielle II

Le cas où $|u| = 0$ suffit souvent :

```
(define F
  (lambda (l)
    (if (null? l)
        c
        (H (F (cdr l))
           l))))
```

Listes : récursivité superficielle III

```
(define length
  (lambda (l)
    (if (null? l) 0 (+ 1 (length (cdr l))))))

(define append
  (lambda (l v)
    (if (null? l) v (cons (car l) (append (cdr l) v)))))

(define reverse
  (lambda (l)
    (if (null? l) '() (append (reverse (cdr l)) (list (car l))))))

(define map
  (lambda (f l)
    (if (null? l) '() (cons (f (car l)) (map f (cdr l))))))
```

Listes : récursivité superficielle IV

```
(define map
  (lambda (f1 l)
    (if (null? l)
        '()
        (cons (f1 (car l)) (map f1 (cdr l))))))
```

s'obtient en instanciant le schéma

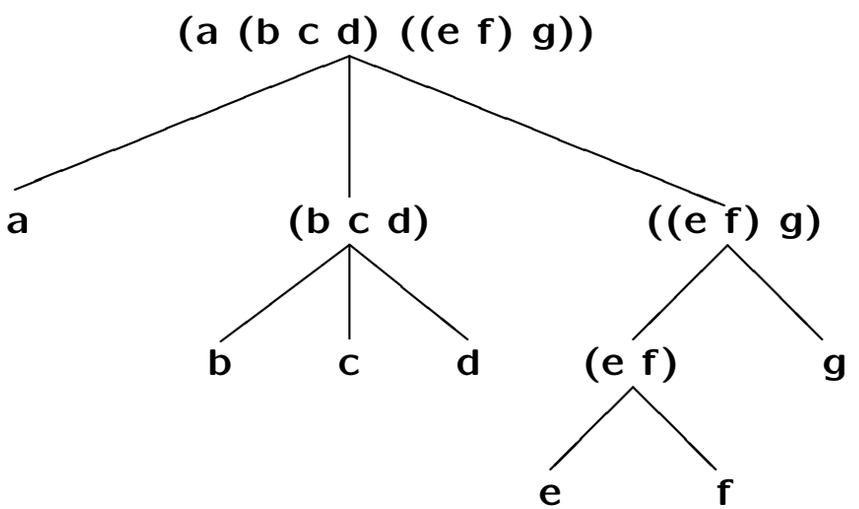
```
(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u))
           l
           u))))

(define G (lambda (u) '()))
(define H (lambda (r l u) (cons (u (car l)) r)))
(define K (lambda (l u) u))
(define map (lambda (f1 l) (F l f1)))
```

Retournement superficiel (“reverse”)

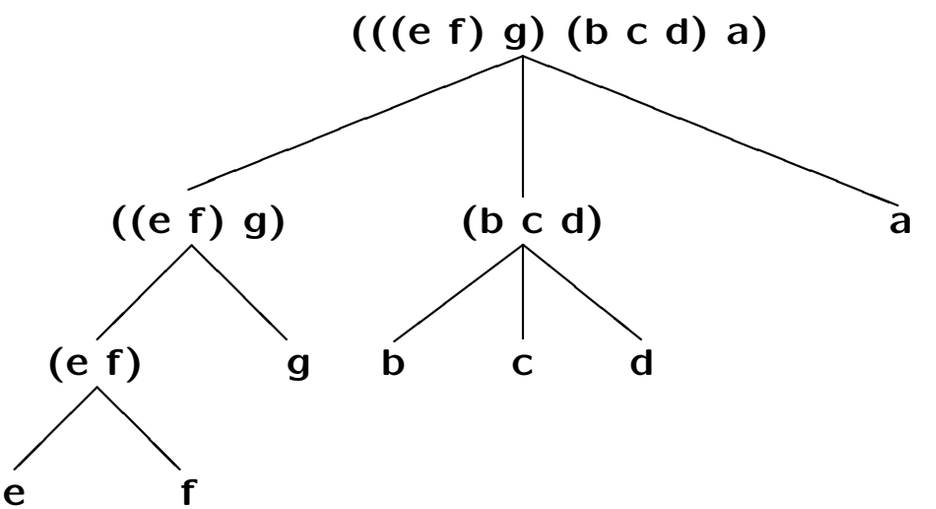
```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l)) (list (car l))))))
```

L'appel récursif porte sur le `cdr` seul.

Argument :



Résultat :



Tri par insertion I

```
(define sort
  (lambda (l comp)   ;; argument procédural
    (if (null? l)
        l
        (insert (car l) (sort (cdr l) comp) comp))))

(define insert
  (lambda (x l comp)
    (cond ((null? l) (list x))
          ((comp x (car l)) (cons x l))
          (else (cons (car l) (insert x (cdr l) comp))))))

(insert 3 '(0 2 3 3 5 7 8 9) <)      (0 2 3 3 3 5 7 8 9)

(sort '(8 3 5 7 2 3 9 0) <=)         (0 2 3 3 5 7 8 9)

(sort '(8 3 5 7 2 3 9 0) >=)         (9 8 7 5 3 3 2 0)
```

Tri par insertion II

```
(define insert
  (lambda (x l comp)
    (if (null? l)
        (list x)
        (if (comp x (car l))
            (cons x l)
            (cons (car l) (insert x (cdr l) comp))))))

(define F
  (lambda (l u1 u2)
    (if (null? l)
        (G u1 u2)
        (H (F (cdr l) (K1 l u1 u2) (K2 l u1 u2)) l u1 u2))))

(define G (lambda (u1 u2) (list u1)))
(define H
  (lambda (r l u1 u2) (if (u2 u1 (car l)) (cons u1 l) (cons (car l) r))))
(define K1 (lambda (l u1 u2) u1))
(define K2 (lambda (l u1 u2) u2))
(define insert (lambda (x l comp) (F l x comp)))
```

Tri par insertion III

```
(define sort
  (lambda (l comp)   ;; argument procédural
    (if (null? l)
        l
        (insert (car l) (sort (cdr l) comp) comp))))

(define F
  (lambda (l u)
    (if (null? l)
        (G u)
        (H (F (cdr l) (K l u)) l u))))

(define G (lambda (u) '()))
(define H (lambda (r l u) (insert (car l) r u)))
(define K (lambda (l u) u))
(define sort (lambda (l comp) (F l comp)))
```

Tri par insertion (ordre lexicographique)

```
(sort '("bb" "ac" "bc" "acb") string<=?)            ("ac" "acb" "bb" "bc")

(define lex
  (lambda (str-ord iden)  ;; arguments procéduraux
    (lambda (u v)
      (cond ((null? u) #t)
            ((null? v) #f)
            ((str-ord (car u) (car v)) #t)
            ((iden (car u) (car v)) ((lex str-ord iden) (cdr u) (cdr v)))
            (else #f)))))

(define numlex (lex < =))
(define alpha (lex string<? string=?))

(sort '((2 3) (1 4) (2 3 2) (1 3)) numlex)    ((1 3) (1 4) (2 3) (2 3 2))

(sort '(("acb" "ac") ("ac" "acb") ("ac")) alpha)
                                        (("ac") ("ac" "acb") ("acb" "ac"))
```

Récursivité profonde sur les listes I

Schéma de base

```
(define F
  (lambda (l u)
    (cond ((null? l) (G u))
          ((atom? (car l))
           (H (F (cdr l) (K l u))
              l
              u))
          ((list? (car l))
           (J (F (car l) (Ka l u))
              (F (cdr l) (Kd l u))
              l
              u)))))
```

Schéma simplifié

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? (car l))
           (H (F (cdr l)) l))
          ((list? (car l))
           (J (F (car l))
              (F (cdr l))
              l)))))
```

Remarque. On exclut (provisoirement) de rencontrer dans la liste `l` des objets qui ne soient ni des listes ni des atomes.

Récursivité profonde sur les listes II

```
(define flat-l
  (lambda (l)
    (cond ((null? l) '())
          ((atom? (car l))
           (if (null? (car l))
               (flat-l (cdr l))
               (cons (car l)
                     (flat-l (cdr l)))))
          ((list? (car l))
           (append (flat-l (car l))
                   (flat-l (cdr l)))))))
```

```
(flat-l 5)                          Error - 5 passed as argument to car

(flat-l '())                        ()

(flat-l '(0 (1 2) ((((3))) 4) 5))   (0 1 2 3 4 5)
```

Récursivité profonde sur les listes III

Les listes ont pour éléments des objets d'un certain type (par exemple, des atomes) et aussi d'autres listes. On peut décider d'adjoindre à ce domaine celui des objets (atomes). Cela revient à admettre les arbres réduits à une racine (qui est en même temps l'unique feuille).
Les schémas précédents s'adaptent facilement.

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? (car l))
           (H (F (cdr l)) l))
          ((list? (car l))
           (J (F (car l))
              (F (cdr l))
              l)))))
```

```
(define F
  (lambda (l)
    (cond ((null? l) c)
          ((atom? l)
           (G l))
          ((list? l)
           (J (F (car l))
              (F (cdr l))
              l)))))
```

Remarque. Le remplacement de `list?` par `pair?` améliore l'efficacité.

Récursivité profonde sur les listes IV

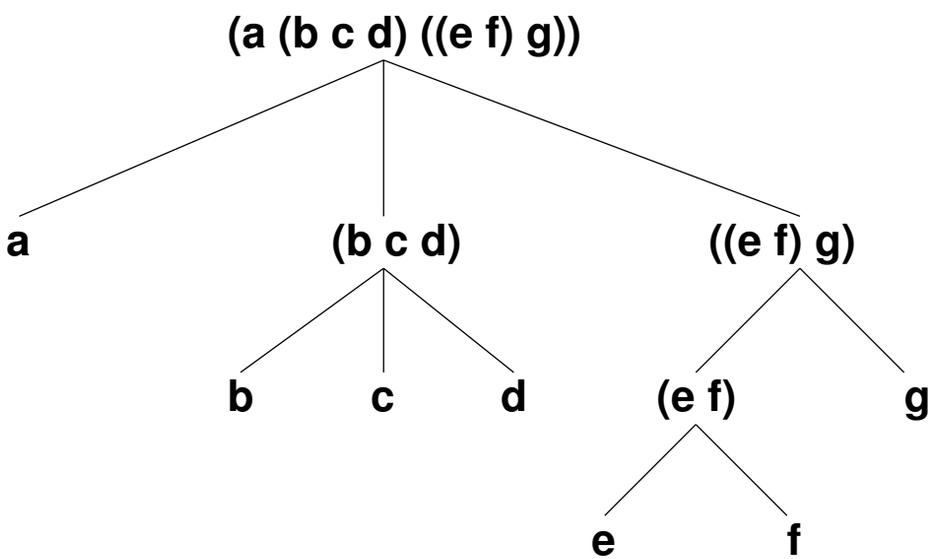
```
(define flat-le
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) (list l))
          ((list? l) (append (flat-le (car l)) (flat-le (cdr l)))))))

(flat-le 5)                              (5)

(flat-le '())                            ()

(flat-le '(0 (1 2) ((((3))) 4) 5))       (0 1 2 3 4 5)
```

Retournement profond (“deeprev”)

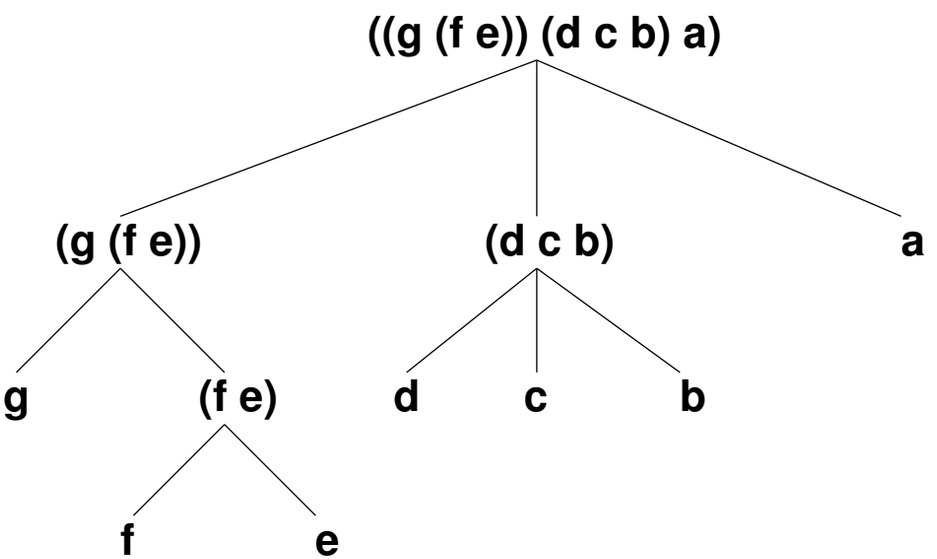
```
(define deeprev
  (lambda (l)
    (cond ((null? l) '())
          ((atom? l) l)
          ((list? l)
           (append (deeprev (cdr l))
                   (list (deeprev (car l))))))))
```

L'appel récursif porte sur le `car` **et** le `cdr`.

Argument



Résultat



Remarque sur les schémas

Suivre un schéma “à la lettre”, c'est-à-dire se limiter strictement à instancier les paramètres qu'il contient, rend la programmation particulièrement méthodique et sûre.
On peut cependant, pour diverses raisons, garder “l'esprit” d'un schéma sans respecter la lettre (sa syntaxe précise). Un exemple classique est le programme `flatten`, version équivalente mais plus efficace de `flat-le` :

```
(define flatten
  (lambda (l u)
    (cond ((null? l) u)
          ((atom? l) (cons l u))
          ((list? l) (flatten (car l) (flatten (cdr l) u))))))
```

On montre facilement que les valeurs de `(flatten l u)` et de `(append (flat-le l) u)` sont égales pour toutes listes `l` et `u` ; en particulier, `(flatten l '())` et `(flat-le l)` ont même valeur. Par contre, `flatten` n'est pas une instance du schéma, même s'il s'en inspire nettement.

On concilie méthode, discipline et créativité . . .

Récursivité structurelle généralisée I

Le principe de la récursivité structurelle est que la structure du programme est calquée sur celle des données.

Exprimer $f(x, \ldots)$ *en termes de* $\{f(y, \ldots) : y \prec x\}$
où $y \prec x$ *signifie* y *composant de* x*.*

On peut généraliser :

- Admettre aussi les composants non immédiats :
  `(- n 1)`, mais aussi `(- n 2)`, `(/ n 2)`, ...
  `(cdr l)`, mais aussi `(cddr l)`, ...
- Admettre plusieurs appels récursifs.
- Admettre les appels imbriqués.

La terminaison reste garantie mais pas l'efficacité !

Récursivité structurelle généralisée II

Attention aux erreurs grossières, telles les évaluations de
`(f (- n 2))` avec `n` ≤ 1, et de
`(f (cddr l))` où `l` comporte moins de deux éléments.

```
(define fib ;; inefficace
  (lambda (n)
     (if (< n 2)
         n
         (+ (fib (- n 1)) (fib (- n 2))))))

(define exp ;; très efficace
  (lambda (m n)
     (cond ((zero? n) 1)
           ((even? n) (exp (* m m) (/ n 2)))
           ((odd? n) (* m (exp m (- n 1)))))))
```

Récursivité structurelle mixte

Le principe de la récursivité structurelle est :
Exprimer $f(x, \ldots)$ *en termes de* $\{f(y, \ldots) : y \prec x\}$ *;*
l'induction porte sur un seul argument.

Le principe de la récursivité structurelle mixte est :
Exprimer $f(x1, x2, \ldots)$ *en termes de*
$\{f(y1, x2, \ldots), f(x1, y2, \ldots), f(y1, y2, \ldots) : y1 \prec x1 \wedge y2 \prec x2\}$.
L'induction porte sur plusieurs arguments ; si $f(a, b)$ *dépend de* $f(c, d)$*,*
alors $c \prec a \wedge d \preceq b$ *ou* $c \preceq a \wedge d \prec b$.
La terminaison reste garantie.

```
(define gcd
  (lambda (x y)
    (cond ((= x y) x)
          ((> x y) (gcd (- x y) y))
          ((< x y) (gcd x (- y x))))))
```

Exemples classiques (voir plus loin) : *knapsack*, *money change*, ...

Séparation fonctionnelle I

On peut "dérécursiver" le schéma classique
$fact(n) :=$ if $n = 0$ then 1 else $n * fact(n-1)$ en l'écriture
$fact_m(n) :=$ if $n = 0$ then 1 else $n * fact_{m-1}(n-1)$.

La récursivité revient (sous forme dégénérée) pour définir globalement les fonctions $fact_m$:

```
(define f
  (lambda (m c)
    (if (zero? m)
        c
        (f (- m 1) (lambda (n) (if (= n 0) 1 (* n (c (- n 1)))))))))
```

Si [[m]] $= m$ et [[c]] $= fact_p$, alors [[(f m c)]] $= fact_{p+m}$

```
(define fact0 'emptyfunction)
(define fact (lambda (n) ((f (+ n 1) fact0) n)))
```

On observe que $fact_m$ a pour domaine $\{0, 1, \ldots, m-1\}$.
Sur ce domaine, on a $fact_m(n) = n!$.

Séparation fonctionnelle II

Exemples

```
(define fact8 (f 8 fact0))

(fact8 7)   5040
(fact8 8)   Error
(fact 8)    40320
```

On a séparé

— le *calcul* de la fonction $fact_p$

— l'*application* de cette fonction à un argument.

On a $fact(n) = fact_p(n)$ si $p > n$; on choisit $p = n + 1$.

Cette technique de *séparation fonctionnelle* n'apporte rien dans le cas très simple de la fonction factorielle mais, dans le cadre général de la définition récursive de fonctions, cette technique sera parfois très utile !

Plus généralement, l'introduction de paramètres fonctionnels et/ou de fonctionnelles auxiliaires définies récursivement est une technique importante.

Séparation fonctionnelle III Processus de calcul

```
(define fact (lambda (n) ((f (+ n 1) fact0) n)))
(define f
  (lambda (m c)
    (if (zero? m)
        c
        (f (- m 1) (lambda (n) (if (= n 0) 1 (* n (c (- n 1)))))))))

(fact 8)
((f 9 fact0) 8)
((f 8 (lambda (n) (if (= n 0) 1 (* n (fact0 (- n 1)))))) 8)
((f 8 fact1) 8)
...
((f 0 fact9) 8)
(fact9 8)
(* 8 (* ... 1))
...
(* 8 5040)
40320
```

L'expansion fonctionnelle précède le calcul arithmétique.

Séparation fonctionnelle IV

Double comptage.

```
(count '(a b a c d a c) 'a)            (3 . 4)
;; 3 occurrences de "a", 4 autres occurrences
```

Solution simple. Utiliser deux fonctions auxiliaires.

```
(define count0 (lambda (l s) (cons (c-eq l s) (c-dis l s))))
```

Problème : on parcourt la liste `l` deux fois.

Solution naïve. Utiliser le schéma habituel.

```
(define count1
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (if (eq? (car l) s)
            (cons (1+ (car (count1 (cdr l) s))) (cdr (count1 (cdr l) s)))
            (cons (car (count1 (cdr l) s)) (1+ (cdr (count1 (cdr l) s))))))))
```

L'inefficacité est catastrophique, mais on peut y remédier simplement.

Séparation fonctionnelle V

La séparation fonctionnelle est une solution possible :

```
(define c2  ;; écrire une spécification !
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l) s (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c2 (cdr l) s (lambda (u) (c (cons (car u) (1+ (cdr u)))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) '(0 . 0))))
```

Si ℓ est la longueur de la liste `l`, les temps d'exécution de `(count0 l)` et de `(count2 l)` sont proportionnels à ℓ ; celui de `(count1 l)` est proportionnel à 2^ℓ.

(On verra d'autres solutions pour ce problème plus loin.)

Séparation fonctionnelle VI

Produit de liste : $\ell \mapsto \prod_{x \in \ell} x$.

```
(define pl1
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (else (* (car l) (pl1 (cdr l)))))))
```

Si un facteur est nul, comment éviter *toutes* les multiplications ?
Ici aussi, la séparation fonctionnelle permet une solution.

```
(define p2  ;; écrire une spécification !
  (lambda (l c)
    (cond ((null? l) c)
          ((zero? (car l)) (lambda (v) 0))
          (else (p2 (cdr l) (lambda (u) (* (car l) (c u))))))))

(define pl2 (lambda (l) ((p2 l id) 1)))
```

On verra une autre solution plus loin.

Séparation fonctionnelle VII

Spécifications de c2 *et de* p2.

```
(define (cx p1) ;; uniquement pour faciliter la spécification
  (lambda (p2) (cons (+ (car p1) (car p2)) (+ (cdr p1) (cdr p2)))))
```

Si `[[(count2 l s)]]` = `[[(cons a b)]]`,
alors `[[(c2 l s (cx (cons u v)))]]` = `[[(cx (cons (+ u a) (+ v b)))]]`

Si `[[l]]` est une liste de nombres dont le produit vaut a,
et si `[[c]]` est la fonction $[x \mapsto bx]$,
alors `[[(p2 l c)]]` est la fonction $[x \mapsto abx]$.

7. Accumulateurs, processus itératifs

Idée de base

Un accumulateur est un argument supplémentaire,
lié à des résultats intermédiaires à mémoriser.

L'exploitation de la définition

```
(define fact (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
```

crée le processus de gauche, alors que l'associativité de la multiplication permettrait celui de droite, plus simple :

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 2))
(* 4 6)
24
```

```
(fact 4)
(* 4 (fact 3))
(* 12 (fact 2))
(* 24 (fact 1))
(* 24 (fact 0))
(* 24 1)
24
```

L'intérêt du processus

```
(* 1 (fact 4))
(* 4 (fact 3))
(* 12 (fact 2))
(* 24 (fact 1))
(* 24 (fact 0))
24
```

est que chaque état est caractérisé par deux paramètres seulement. On peut générer un tel processus très simplement, en faisant de ces paramètres les arguments d'une fonction `fact-a` :

```
(fact-a 4 1)
(fact-a 3 4)
(fact-a 2 12)
(fact-a 1 24)
(fact-a 0 24)
24
```

La définition de `fact-a` est évidente :

```
(define fact-a
  (lambda (n a)
    (if (zero? n) a (fact-a (- n 1) (* a n)))))
```

Si [[n]] $= n$ et [[a]] $= a$,
alors `[[(fact-a n a)]]` $= n!a$.

Le processus généré est analogue à celui associé à la boucle

```
while n > 0 do (a,n) := (a*n,n-1)
```

Processus itératif : espace-mémoire de contrôle constant.

```
(define fact (lambda (n) (fact-a n 1)))
```

Variante.

On observe qu'au cours de l'exécution l'accumulateur a pour valeur un produit partiel, tel $4 * 3$ ou $4 * 3 * 2$ (si on calcule 4!). Rien n'empêche d'utiliser plutôt des produits partiels du type $1 * 2$ ou $1 * 2 * 3$.

On définit `(fact-b n i b)` par la spécification suivante :

> Si [[n]] $= n$, [[i]] $= i$ et [[b]] $= (i-1)!$ et si $1 \leq i \leq n+1$,
> alors [[`(fact-b n i b)`]] $= n!$.

On peut écrire :

```
(define fact-b
  (lambda (n i b)
    (if (> i n) b (fact-b n (+ i 1) (* b i)))))

(define fact
  (lambda (n) (fact-b n 1 1)))
```

Un exemple classique

```
(define fib
  (lambda (n)
    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))
```

Problème : recalcul inutile de résultats intermédiaires.

Solution : utiliser des accumulateurs.

```
(define fib-a
  (lambda (n a b)
    (if (zero? n)
        a
        (fib-a (- n 1) b (+ a b)))))
```

Processus itératif, comme pour la boucle

`while n > 0 do (n,a,b) := (n-1,b,a+b)`

```
(fib-a  9  0  1)
(fib-a  8  1  1)
(fib-a  7  1  2)
(fib-a  6  2  3)
(fib-a  5  3  5)
(fib-a  4  5  8)
(fib-a  3  8 13)
(fib-a  2 13 21)
(fib-a  1 21 34)
(fib-a  0 34 55)
34
```

Autres exemples numériques I

```
(define expt
 (lambda (m n)
  (cond ((zero? n) 1)
        ((even? n) (expt (* m m) (/ n 2)))
        ((odd? n) (* m (expt m (- n 1)))))))

(define expt-a
 (lambda (m n a)
  (cond ((zero? n) a)
        ((even? n) (expt-a (* m m) (/ n 2) a))
        ((odd? n) (expt-a m (- n 1) (* m a))))))
```

Si [[m]] $= m$, [[n]] $= n$ et [[a]] $= a$ alors [[(expt-a m n a)]] $= m^n * a$.

```
(expt 5 4)              625
(expt-a 5 4 10)        6250
```

Autres exemples numériques II

```
(define cbin
 (lambda (n u)
  (if (zero? n)
      1
      (/ (* (cbin (- n 1) (- u 1)) u) n))))

(define cbin-a  ;; à spécifier
 (lambda (n u a)
  (if (zero? n)
      a
      (cbin-a (- n 1) (- u 1) (/ (* u a) n)))))

(cbin 4 6)              15
(cbin-a 4 6 1)          15
(cbin-a 4 6 10)        150
```

Exemple avec exploration superficielle I

```
(define reverse
  (lambda (u)
    (if (null? u)
        '()
        (append (reverse (cdr u))
                (list (car u))))))

(reverse '(0 1 2))
(append (reverse '(1 2)) '(0))
(append (append (reverse '(2)) '(1)) '(0))
(append (append (append (reverse '()) '(2)) '(1)) '(0))
(append (append (append '() '(2)) '(1)) '(0))
(append (append '(2) '(1)) '(0))
(append '(2 1)) '(0))                              (2 1 0)
```

Temps d'exécution quadratique

Exemple avec exploration superficielle II

```
(define reverse-a
  (lambda (u a)
    (if (null? u)
        a
        (reverse-a (cdr u) (cons (car u) a)))))
```

Temps d'exécution linéaire, processus itératif

```
(reverse-a '(0 1 2) '())
(reverse-a '(1 2) '(0))
(reverse-a '(2) '(1 0))
(reverse-a '() '(2 1 0))                    (2 1 0)
```

[[(append (reverse u) a)]] = [[(reverse-a u a)]]

Exemple avec exploration profonde

```
(define flat-l
  (lambda (l)
    (cond
      ((null? l) '())
      ((atom? (car l))
       (if (null? (car l)) (flat-l (cdr l)) (cons (car l) (flat-l (cdr l)))))
      ((list? (car l)) (append (flat-l (car l)) (flat-l (cdr l)))))))
```

```
(define flat-a
  (lambda (l a)
    (cond
      ((null? l) a)
      ((atom? (car l))
       (if (null? (car l))
           (flat-a (cdr l) a)
           (cons (car l) (flat-a (cdr l) a))))
      ((list? (car l)) (flat-a (car l) (flat-a (cdr l) a))))))
```

`[[(append (flat-l l) a)]]` = `[[(flat-a l a)]]`

Gain de temps et d'espace, processus *non* itératif.

Exemple, récursivité non structurelle I

```
(define M (lambda (x) (if (> x 100) (- x 10) (M (M (+ x 11))))))
```

On peut démontrer que la valeur de `(M x)` existe
quelle que soit la valeur de l'entier (relatif) `x`.

On peut aussi prouver que le graphe de `M` coïncide avec celui de `M91` :

```
(define M91 (lambda (x) (if (> x 100) (- x 10) 91)))
```

```
[[(M-c x 0)]] = [[x]];
[[(M-c x 1)]] = [[(M x)]];
[[(M-c x 2)]] = [[(M (M x))]];
[[(M-c x 3)]] = [[(M (M (M x)))]];
...
```

```
(define M-c
  (lambda (x c)
    (cond ((= c 0) x)
          ((> x 100) (M-c (- x 10) (- c 1)))
          (else (M-c (+ x 11) (+ c 1))))))
```

Exemple, récursivité non structurelle II

```
(define ack   ;; Fonction totale sur NxN
  (lambda (m n)
    (cond
      ((zero? m) (+ n 1))
      ((zero? n) (ack (- m 1) 1))
      (else (ack (- m 1) (ack m (- n 1)))))))

(define acl
  (lambda (l)
    (cond
      ((null? (cdr l)) (car l))
      ((zero? (cadr l)) (acl (cons (+ (car l) 1) (cddr l))))
      ((zero? (car l)) (acl (cons 1 (cons (- (cadr l) 1) (cddr l)))))
      (else (acl (cons (- (car l) 1) (cons (cadr l) (cons (- (cadr l) 1) (cddr l)))))))))
```

```
[[(acl '(n))]] = [[n]];
[[(acl '(n m))]] = [[(ack m n)]];
[[(acl '(n m u))]] = [[(ack u (ack m n))]];
[[(acl '(n m u v))]] = [[(ack v (ack u (ack m n)))]];
...
```

Programmation CPS I

Reconsidérons deux états homologues des processus de calcul associés aux évaluations des formes `(fact 4)` et `(fact-a 4 1)`, soient

```
(* 4 (* 3 (fact 2)))        (fact-a 2 12)
```

On a trois paires de constituants homologues :

```
fact                        fact-a
2                           2
(* 4 (* 3 ...))            12
```

Il est plus économique de mémoriser le nombre `12` que la fonction `(* 4 (* 3 ...))` ou, plus exactement, l'expression `(lambda (k) (* 4 (* 3 k)))`.

Programmation CPS II

On peut cependant utiliser un argument fonctionnel, et on le devrait si la multiplication n'était pas associative :

```
(define fact-c
  (lambda (n c)
    (if (zero? n)
        (c 1)
        (fact-c (- n 1)
                (lambda (k) (c (* n k)))))))

(define fact
  (lambda (n) (fact-c n (lambda (k) k))))
```

Si n est un naturel
et si c est une fonction de $\mathbb{N}$ dans D,
alors [[(fact-c n c)]] $=$ $c(n!)$.

Programmation CPS III

On voit que l'associativité n'est pas utilisée :

```
(fact-c 4 (lambda (k) k))
(fact-c 3 (lambda (k) ((lambda (k) k) (* 4 k))))
(fact-c 3 (lambda (k) (* 4 k)))
(fact-c 2 (lambda (k) ((lambda (k) (* 4 k)) (* 3 k))))
(fact-c 2 (lambda (k) (* 4 (* 3 k))))
...
(fact-c 0 (lambda (k) (* 4 (* 3 (* 2 (* 1 k))))))
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
(* 4 (* 3 (* 2 (* 1 1))))
24
```

Programmation CPS IV

Remarque. Scheme n'effectue les réductions qu'à la fin du processus. La dernière étape n'est donc pas l'évaluation de

```
((lambda (k) (* 4 (* 3 (* 2 (* 1 k))))) 1)
```

mais celle de

```
((lambda (k)
   ((lambda (k)
      ((lambda (k)
         ((lambda (k)
            ((lambda (k) k) (* 4 k)))
          (* 3 k)))
       (* 2 k)))
    (* 1 k)))
 1)
```

Programmation CPS V

L'argument fonctionnel auxiliaire `c` est une *continuation*. Cette fonction, appliquée à un résultat intermédiaire du calcul en cours, fournit le résultat final.

Dans le cas de `fact-c`, l'exécution construit itérativement une fonction de plus en plus complexe, représentant l'enchaînement des multiplications à faire. A chaque étape, l'argument continuation est transformé en une fonction impliquant une multiplication supplémentaire ; cette fonction est l'argument de l'appel suivant. Les calculs numériques n'ont lieu que quand l'enchaînement complet des multiplications a été formé. Cette technique est le *Continuation-Passing Style* (CPS). Elle ressemble à la technique de séparation fonctionnelle vue précédemment.

Programmation CPS VI

```
(define pl
  (lambda (l)
    (cond ((null? l) 1)
          ((zero? (car l)) 0)
          (#t (* (car l) (pl (cdr l)))))))
```

Produit de liste : $\ell \mapsto \prod_{x \in \ell} x$.

Si un facteur est nul, comment éviter *toutes* les multiplications ?
Séparation fonctionnelle, ou encore CPS :

```
(define pl-c
  (lambda (l c)
    (cond ((null? l) (c 1))
          ((zero? (car l)) 0)
          (#t (pl-c (cdr l) (lambda (k) (c (* (car l) k))))))))

(define pl (lambda (l) (pl-c l (lambda (k) k))))
```

On comparera les couples (`pl-c` , `pl`) et (`p2` , `pl2`), tr. 86.

Programmation CPS VII

On peut remplacer la continuation par un argument non fonctionnel. Pour `pl`, cela revient à tester l'absence de facteur nul avant d'opérer les multiplications.

```
(define pl-l
  (lambda (l a) ;; la liste a ne comporte pas de 0
    (cond ((null? l) (p a))
          ((zero? (car l)) 0)
          (else (pl-l (cdr l) (cons (car l) a))))))

(define pl (lambda (l) (pl-l l '())))

;; p est une procedure calculant le produit
;; d'une liste dont aucun facteur n'est nul.
```

Inconvénient : double parcours de liste si aucun facteur nul n'est présent.

Programmation CPS VIII

Double comptage – rappel. La séparation fonctionnelle permet d'éviter efficacement le double parcours de la liste :

```
(define c2
  (lambda (l s c)
    (if (null? l)
        c
        (if (eq? (car l) s)
            (c2 (cdr l)
                s
                (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c2 (cdr l)
                s
                (lambda (u) (c (cons (car u) (1+ (cdr u))))))))))

(define id (lambda (v) v))

(define count2 (lambda (l s) ((c2 l s id) '(0 . 0))))
```

Programmation CPS IX

La solution en CPS est semblable :

```
(define c3
  (lambda (l s c)
    (if (null? l)
        (c '(0 . 0))
        (if (eq? (car l) s)
            (c3 (cdr l)
                s
                (lambda (u) (c (cons (1+ (car u)) (cdr u)))))
            (c3 (cdr l)
                s
                (lambda (u) (c (cons (car u) (1+ (cdr u))))))))))

(define count3 (lambda (l s) (c3 l s id)))
```

Programmation CPS X

Une variante permet l'économie des opérations sur les paires :

```
(define c3
  (lambda (l s c)
    (if (null? l)
        (c 0 0)
        (if (eq? (car l) s)
            (c3 (cdr l)
                s
                (lambda (u v) (c (1+ u) v)))
            (c3 (cdr l)
                s
                (lambda (u v) (c u (1+ v))))))))

(define count3 (lambda (l s) (c3 l s cons)))
```

Programmation CPS XI

Spécification :

Si s est un symbole,
si ℓ est une liste de symboles
et si c est une fonction de $\mathbb{N} \times \mathbb{N}$ dans D,
alors (c3 ℓ s c) vaut $c(a, b)$,
où a est le nombre d'occurrences de s dans ℓ
et où b est le nombre d'occurrences
de symboles distincts de s dans ℓ.

Programmation CPS XII

L'argument fonctionnel évolue comme suit.

Si sa valeur "avant" un appel récursif est celle de `c`, soit

```
(lambda (x y) (c x y))
```

sa "nouvelle" valeur sera celle d'une des expressions

```
(lambda (x y) (c (1+ x) y))
(lambda (x y) (c x (1+ y)))
```

Programmation CPS XIII

On peut remplacer l'argument fonctionnel par deux arguments numériques. Cela donne :

```
(define c4
  (lambda (l s a1 a2)
    (if (null? l)
        (cons a1 a2)
        (if (eq? (car l) s)
            (c4 (cdr l) s (1+ a1) a2)
            (c4 (cdr l) s a1 (1+ a2))))))

(define count4 (lambda (l s) (c4 l s 0 0)))
```

Cette solution est simple et optimale.

On voit que la séparation fonctionnelle et le CPS sont des techniques utiles, donnant lieu à des solutions efficaces.

Parfois, le remplacement de l'argument fonctionnel par un ou plusieurs accumulateur(s) améliore encore le programme.

Un schéma accumulant I

Si à toute liste de naturels la fonction h associe un naturel, on définit la fonction gib (pour "generalized Fibonacci") par

$$gib(n, h) \;=\; h([gib(n-1, h), \ldots, gib(0, h)])\,.$$

Pour une fonction h appropriée, on retrouve Fibonacci :

```
(define hfib
  (lambda (l)
    (cond ((null? l) 0)
          ((null? (cdr l)) 1)
          (else (+ (car l) (cadr l))))))

(hfib '(8 5 3 2 1 1 0))   13
```

Un schéma accumulant II

Cas général :

```
(define rev-enum        ;; 5 -> (5 4 3 2 1 0)
  (lambda (n)
    (if (< n 0) '() (cons n (rev-enum (- n 1))))))
```

on peut programmer *gib* en utilisant `map` :

```
(define gib0
  (lambda (n h)
    (h (map (lambda (i) (gib0 i h))
            (rev-enum (- n 1))))))

(gib0 12 hfib)   144
```

C'est correct, mais extrêmement inefficace.

Un schéma accumulant III

En utilisant une fonction auxiliaire, on obtient :

```
(define gib1
  (lambda (n h) (h (gib1* n h))))

(define gib1*
  (lambda (n h)
    (if (= n 0)
        '()
        (cons (gib1 (- n 1) h) (gib1* (- n 1) h)))))
```

Cette solution reste inefficace car l'évaluation du premier argument du `cons` implique la (ré)évaluation du second. Plus précisément, si la valeur du second argument est r, la valeur du premier argument est $h(r)$.

```
(gib1 10 hfib)   55

(gib1* 10 hfib)  (34 21 13 8 5 3 2 1 1 0)

(hfib '(34 21 13 8 5 3 2 1 1 0))   55
```

Un schéma accumulant IV

La variante suivante évite ce gaspillage :

```
(define gib2 (lambda (n h) (h (gib2* n h))))

(define gib2*
  (lambda (n h)
    (if (= n 0)
        '()
        ((lambda (rec) (cons (h rec) rec)) (gib2* (- n 1) h)))))
```

La fonction auxiliaire s'écrit aussi (variante syntaxique vue plus loin)

```
(define gib2*
  (lambda (n h)
    (if (= n 0)
        '()
        (let ((rec (gib2* (- n 1) h))) (cons (h rec) rec)))))
```

On a par exemple

```
(gib2* 12 hfib)                    (89 55 34 21 13 8 5 3 2 1 1 0)
```

Un schéma accumulant V

Version accumulante et itérative :

```
(define gib (lambda (n h) (h (gib-a n h '()))))

(define gib-a
  (lambda (n h l)
    (if (= n 0) l (gib-a (- n 1) h (cons (h l) l)))))
```

Si `n`, `h` et `l` ont pour valeurs respectives
n, h et $[gib(i-1,h),\ldots,gib(0,h)]$,
alors `(gib-a n h l)` a pour valeur
$[gib(n+i-1,h),\ldots,gib(0,h)]$.
En particulier, `(gib-a n h '())` a pour valeur
$[gib(n-1,h),\ldots,gib(0,h)]$.

```
(gib-a 4 hfib '(21 13 8 5 3 2 1 1 0))   (144 89 55 34 21 13 8 5 3 2 1 1 0)
(gib 12 hfib)                           144
```

8. Expressions symboliques

Représentation des listes en mémoire

Principe. La représentation en mémoire de la valeur de (`cons` α β) est un couple de pointeurs vers les représentations des valeurs de α et β. Dans le cas des listes, β est une liste, mais le cas où β n'est pas une liste est admis aussi.

Extension. Une *expression symbolique* est un atome ou une paire formée d'expressions symboliques.

Notation pointée. Le point (entouré d'espaces) et les parenthèses représentent l'appariement.

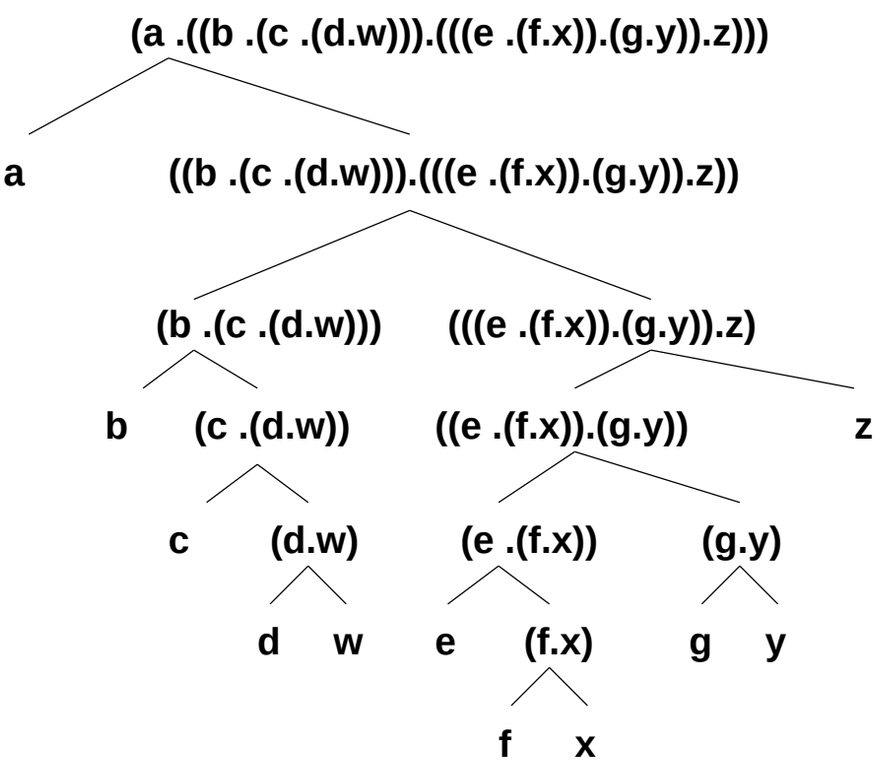
```
'a                --->    a
(cons 'a 'b)      --->    (a . b)
(cons 'a '())     --->    (a . ())
(list a)          --->    (a . ())
(list a b)        --->    (a . (b . ()))
'(a b c d)        --->    (a . (b . (c . (d . ()))))
'((a b) (c))      --->    ((a . (b . ())) . ((c . ()) . ()))
```

Expressions symboliques représentées par des arbres binaires

Une expression symbolique est un arbre binaire dont les feuilles sont des atomes.

```
(a . ((b . (c . (d . w))) . (((e . (f . x)) . (g . y)) . z)))
```

```
(cons a
      (cons (cons b (cons c (cons d w)))
            (cons (cons (cons e (cons f x))
                        (cons g y))
                  z)))
```



Notation pointée et notation usuelle

La notation pointée met en évidence la structure
d'*arbre binaire décoré* des expressions symboliques.
Chaque nœud a 0 (feuille) ou 2 (nœud interne) fils,
auxquels on accède par `car` et `cdr`.
Chaque feuille est (étiquetée par) un atome.

```
a    ()    (b . 3)    ((a . b) . c)    ((7 . g) . (#f . (y . (z . ()))))
```

Les listes sont des expressions symboliques particulières ;
chaque point est suivi d'une parenthèse ouverte :

```
()                  ()
(0)                 (0 . ())
(0 1)               (0 . (1 . ()))
(0 1 2)             (0 . (1 . (2 . ())))
((0 1) 2)           ((0 . (1 . ())) . (2 . ()))
```

Les constructeur et accesseurs et reconnaisseur sont `cons`, `car`, `cdr`, `pair?`.

De la notation pointée à la notation usuelle

Tout point suivi d'une parenthèse ouverte est supprimé, ainsi que la parenthèse ouverte et la parenthèse fermée correspondante.
L'ordre des suppressions est quelconque.
Un point non suivi d'une parenthèse ouverte n'est pas supprimable !

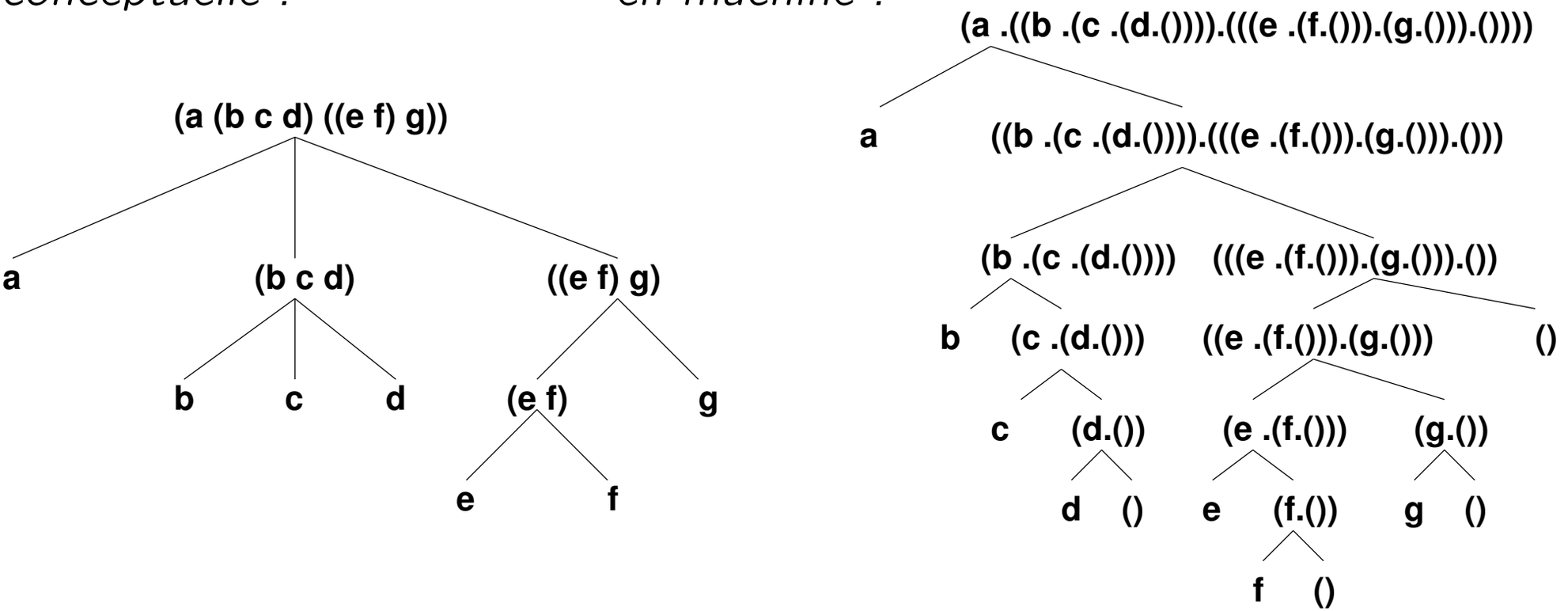
```
((0 . (1 . ())) . (2 . ()))
((0 . (1 . ())) . (2    ))
((0    1 . () ) . (2    ))
((0    1 . () )    2     )
((0    1      )    2     )

((a . (b . ())) . ((c . (d . ())) . ()))
((a . (b . ())) . ((c . (d . (()))))
((a . (b . ())) . ((c . (d))))
((a . (b . ())) . ((c d)))
((a . (b . ())) (c d))
((a . (b)) (c d))
((a b) (c d))

((a . b) . (c . d))
((a . b) c . d)
```

Représentations des listes

Une liste se représente conceptuellement par un arbre (quelconque). En machine, on représente plutôt (par un arbre binaire) l'expression symbolique équivalente (en fait, égale).

Représentations de la liste `(a (b c d) ((e f) g))`

conceptuelle : *en machine :*



Remarque. L'information attachée à un nœud interne se déduit de celle attachée à ses descendants ; seule l'information attachée aux feuilles est explicitement représentée.

Récursivité structurelle : les expressions symboliques

Schéma de base

```
(define F
  (lambda (s u)
    (if (atom? s)
        (G s u)
        (H (F (car s) (Ka s u))
           (F (cdr s) (Kd s u))
           s
           u))))
```

Schéma simplifié

```
(define F
  (lambda (s)
    (if (atom? s)
        (G s)
        (H (F (car s))
           (F (cdr s))
           s))))
```

Récursivité structurelle : exemples symboliques

```
(define size
  (lambda (s)
    (if (atom? s) 1 (+ (size (car s)) (size (cdr s))))))

(define flatten
  (lambda (s)
    (if (atom? s) (list s) (append (flatten (car s)) (flatten (cdr s))))))

(define flatten-a
  (lambda (s a)
    (if (atom? s) (cons s a) (flatten-a (car s) (flatten-a (cdr s) a)))))

(flatten-a '((a .(b .())).((c .(d .())).())) '(1 2)) (a b () c d () () 1 2)

(flatten-a '((a b) (c d)) '(1 2))                    (a b () c d () () 1 2)

(flat-a '((a b) (c d)) '(1 2))                       (a b c d 1 2)
```

Récursivité structurelle : listes et expressions symboliques

Une expression symbolique peut-elle s'écrire sans point ?

```
(define point-free?
  (lambda (s)
    (or (atom? s)
        (and (point-free? (car s))
             (list? (cdr s))
             (point-free? (cdr s))))))

(point-free? 'a)                    #t
(point-free? '((a . b)))            #f
(point-free? '(a . (b . ())))       #t
(point-free? '(a b . c))            #f
```

Egalité, identité

Egalité : `(equal? x y)` est

```
(cond ((pair? x)
       (and (pair? y) (equal? (car x) (car y)) (equal? (cdr x) (cdr y))))
      ((pair? y) #f)
      (else (eqv? x y))))
```

```
(equal? '(a . (b . c)) '(a b . c))   #t
```

`eq?`	`=`	`eqv?`	`equal?`
`symbol?`	`number?`	`symbol? number?`	`symbol? number? pair?`

Identité. `(eq? x y)` : les valeurs de `x` et `y` sont le même objet en mémoire.

```
(define x '(a . b))   (define y '(a . b))   (define z x)
(eq? 'a 'a)                  #t
(eq? x y)                    #f
(eq? x z)                    #t
```

“Déconstruction”

```
(define describe
  (lambda (s)
    (cond ((null? s) (quote '()))
          ((number? s) s)
          ((symbol? s) (list 'quote s))
          ((pair? s) (list 'cons (describe (car s)) (describe (cdr s))))
          (else s))))

(describe '(1 ((a) 2)))
(cons 1 (cons (cons (cons 'a '())
                    (cons 2 '())) '()))

(cons 1 (cons (cons (cons 'a '())
                (cons 2 '())) '()))
(1 ((a) 2))

(describe '((a . b) . (c . d)))
(cons (cons 'a 'b) (cons 'c 'd))

(cons (cons 'a 'b) (cons 'c 'd))
((a . b) c . d)
```

“Déconstruction – pretty printing”

```
(define lcons 6) ;; longueur de "(cons "

(define dis      ;; (dis n) ecrit n blancs
  (lambda (n)
    (if (zero? n)
        (display "")
        (begin (display " ") (dis (- n 1))))))

(define disp (lambda (n m) (dis (+ n (* m lcons)))))

(define pretty
  (lambda (d n m)
    (if (not (pair? d))
        (begin (disp n 0)
               (if (symbol? d) (display "'"))
               (display d))
        (begin (display "(cons ")
               (pretty (car d) 0 (+ m 1))
               (newline)
               (disp n (+ m 1))
               (pretty (cdr d) 0 (+ m 1))
               (display ")")))))

(define pr (lambda (d) (begin (newline) (pretty d 0 0))))
```

“Déconstruction” – exemples

```
(pr 'a)
'a

(pr '(a . (b . 1)))
(cons 'a
      (cons 'b
            1))

(pr '(a b c d))
(cons 'a
      (cons 'b
            (cons 'c
                  (cons 'd
                        ()))))

(pr '((a . (b . 1)) . ((c . 2) . ((d . 3) . e))))
(cons (cons 'a
            (cons 'b
                  1))
      (cons (cons 'c
                  2)
            (cons (cons 'd
                        3)
                  'e)))
```

9. Abstraction et blocs

Forme spéciale `let` I

Abstraire (nommer) une sous-expression

Calcul de $2(a+b)^2 + (a+b)(a-c)^2 + (a-c)^3$

Approche naïve :

```
(+ (* 2 (+ a b) (+ a b))
   (* (+ a b) (- a c) (- a c))
   (* (- a c) (- a c) (- a c)))
```

Approche économique et structurée :

```
(let ((x (+ a b)) (y (- a c)))
  (+ (* 2 x x)
     (* x y y)
     (* y y y)))
```

Forme spéciale `let` II

Evaluer $2(a+b)^2 + (a+b)(a-c)^2 + (a-c)^3$
en calculant d'abord $x = a+b$ et $y = a-c$,
c'est appliquer la fonction $(x,y) \mapsto 2x^2 + xy^2 + y^3$
aux arguments $x = a+b$ et $y = a-c$.

Définition

`(let ((x` α`) (y` β`))` γ`)` variante syntaxique de `((lambda (x y)` γ`)` α β`)`

```
(let ((x (+ a b))
      (y (- a c)))
  (+ (* 2 x x) (* x y y) (* y y y)))
```

est donc strictement équivalent à

```
((lambda (x y)
   (+ (* 2 x x) (* x y y) (* y y y)))
 (+ a b)
 (- a c))
```

Procédures locales

```
(define hypo
  (lambda (x y)
    (sqrt (+ (square x) (square y)))))

(define square
  (lambda (x) (* x x)))
```

Comment empêcher l'usage autonome de `square` ?

```
(define hypo1
  (let ((square (lambda (x) (* x x))))
    (lambda (x y)
      (sqrt (+ (square x) (square y))))))
```

Variante

```
(define hypo2
  (lambda (x y)
    (let ((square (lambda (x) (* x x))))
      (sqrt (+ (square x) (square y))))))
```

Forme `let*` : abrège des `let` imbriqués

```
(let* ((x (+ a b)) (y (- a b)) (z (* x y)))
 (+ (* x x) z z (* y y)))

(let ((x (+ a b)) (y (- a b)))
  (let ((z (* x y)))
    (+ (* x x) z z (* y y))))

(let ((x (+ a b)))
  (let ((y (- a b)))
    (let ((z (* x y)))
      (+ (* x x) z z (* y y)))))
```

Ces trois formes sont équivalentes ; la valeur dépend des valeurs de `a` et `b`.

par contre, la valeur de

```
(let ((x (+ a b)) (y (- a b)) (z (* x y)))
  (+ (* x x) z z (* y y))))
```

dépend des valeurs de `a`, `b`, `x` et `y`.

Une fonction arithmétique I

$$f(n) =_{def} \left(\sum_{i=0}^{n-1} ([2 + f(i)] * [3 + f(n-i-1)]) \right) \bmod (2n+3)\,.$$

Version naïve, traduction littérale

```
(define f0
  (lambda (n)
    (modulo (apply +
                   (map (lambda (i)
                          (* (+ 2 (f0 i))
                             (+ 3 (f0 (- n i 1)))))
                        (enum 0 (- n 1))))
            (+ n n 3))))
```

```
(time (f0 12))  cpu time:  324           1
(time (f0 13))  cpu time:  964           8        Catastrophique!
(time (f0 14))  cpu time: 2937          16
```

Une fonction arithmétique II

```
(define gib (lambda (n h) (h (gib-a n h '()))))

(define gib-a
  (lambda (n h l) (if (= n 0) l (gib-a (- n 1) h (cons (h l) l)))))

(define harith
  (lambda (l)
    (modulo (apply +
                   (map (lambda (u v) (* (+ 2 u) (+ 3 v)))
                        l
                        (reverse l)))
            (let ((s (length l))) (+ s s 3)))))

(define f1 (lambda (n) (gib n harith)))
```

Une fonction arithmétique III

```
(define f (lambda (n) (fa n 0 '() '())))

(define fa
  (lambda (k i u v)
    (let ((next
           (modulo (apply +
                          (map (lambda (x y)
                                 (* (+ 2 x) (+ 3 y)))
                               u
                               v))
                   (+ i i 3))))
      (if (zero? k)
          next
          (fa (- k 1)
              (+ i 1)
              (cons next u)
              (append v (list next)))))))
```

Une fonction arithmétique IV

Si `k`, `i`, `u` et `v` ont pour valeurs respectives
les naturels i, k et les listes $[f(i-1), ..., f(0)]$ et $[f(0), ..., f(i-1)]$,
alors `(fa k i u v)` a pour valeur $f(k+i)$.

```
(fa 8 5 '(10 3 3 1 0) '(0 1 3 3 10))          8
(f 13)                                        8

(time (f1   14))  cpu time:    0             16
(time (f1  140))  cpu time:    1            195
(time (f1 1400))  cpu time:  206           1477
(time (f  1400))  cpu time:  198           1477
```

Forme `letrec`

`letrec` construction essentielle !

```
(define power4
  (lambda (n)
    (let ((square (lambda (m) (* m m))))
      (square (square n)))))

(define fact
  (lambda (n)
    (letrec ((fact-it
              (lambda (k acc)
                (if (= k 0) acc (fact-it (- k 1) (* k acc))))))
      (fact-it n 1))))
```

`let` permet des définitions locales non récursives seulement ;
`letrec` permet en plus des définitions locales récursives.

Portée I

Les principes de portée et de renommage relatifs à la forme `lambda` restent valables pour `let`, `let*` et `letrec`.

```
(define a 5)                ...
(add1 a)                    6
(let ((a 3)) (add1 a))      4
(let ((c 3)) (add1 c))      4
(add1 3)                    4
(add1 a)                    6
```

```
(define f
  (let ((b 100))
    (lambda (x) (+ x b))))   ...
(let ((b 10)) (f 25))        125
```

La plupart des confusions éventuelles proviennent d'un téléscopage de noms. Il suffit d'appliquer méthodiquement les règles d'évaluation pour éviter les erreurs. On peut aussi renommer (mentalement) certaines variables liées.

Portée II

```
(let ((a 5))
  (let ((fun (lambda (x) (max x a))))
    (let ((a 10) (x 20))
      (fun 1))))                                5

(let ((c 5))
  (let ((fun (lambda (y) (max y c))))
    (let ((a 10) (x 20))
      (fun 1))))                                5

(let ((fun (lambda (x) (max x 5))))
  (let ((a 10) (x 20))
    (fun 1)))                                   5

(let ((fun (lambda (x) (max x 5))))
  (fun 1))                                      5

((lambda (x) (max x 5)) 1)                      5

(max 1 5)                                       5
```

Schémas récursifs avec `let` I

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (H (F (- n 1) (K n u)) n u))))
```

Seule exigence pour `G`, `H` et `K` : la terminaison.
`H` peut être *difficile à découvrir* (c'est rarement le cas pour `G` et `K`).
`H` peut induire un *calcul multiple d'argument* (pas `G` ni `K`).
On réduit significativement le premier problème . . .
et on élimine le second en utilisant `let` :

```
(define F
  (lambda (n u)
    (if (zero? n)
        (G u)
        (let ((v (F (- n 1) (K n u))))
          (H v n u)))))   ;; utile si plusieurs occurrences de v
```

Schémas récursifs avec `let` II

La variante avec `let` est spécialement utile si `(H v n u)` est une expression complexe, comportant plusieurs occurrences de `v`.

```
(define f
  (lambda (n u)
    (if (zero? n)
        u
        (let ((v (f (- n 1) u)))
          (/ (- 2 v (/ (* v v) u)) 3)))))

(define f
  (lambda (n u)
    (if (zero? n)
        u
        (/ (- 2 (f (- n 1) u)
               (/ (* (f (- n 1) u)
                     (f (- n 1) u)) u))
           3)))))
```

Le premier programme est d'efficacité linéaire en `n`,
tandis que le second, moins lisible, est exponentiel.

Schémas récursifs avec `let` III

Double comptage – rappel.

```
(count '(a b a c d a c) 'a)   (3 . 4)
```

L'utilisation brutale du schéma donne un programme inefficace :

```
(define count1
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (if (eq? (car l) s)
            (cons (1+ (car (count1 (cdr l) s)))
                  (cdr (count1 (cdr l) s)))
            (cons (car (count1 (cdr l) s))
                  (1+ (cdr (count1 (cdr l) s))))))))
```

Schémas récursifs avec `let` IV

Une solution efficace est possible avec `let` :

```
(define count5
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (let ((rec (count5 (cdr l) s)))
          (if (eq? (car l) s)
              (cons (1+ (car rec))
                    (cdr rec))
              (cons (car rec)
                    (1+ (cdr rec))))))))
```

Schémas récursifs avec `let` V

La solution est strictement équivalente à ou à

```
(define count5
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        ((lambda (rec)
           (if (eq? (car l) s)
               (cons (1+ (car rec))
                     (cdr rec))
               (cons (car rec)
                     (1+ (cdr rec)))))
         (count5 (cdr l) s)))))
```

```
(define count5
  (lambda (l s)
    (if (null? l)
        (cons 0 0)
        (aux (count5 (cdr l) s)))))

(define aux
  (lambda (rec)
    (if ... (1+ (cdr rec)))))
```

On comparera cette solution aux précédentes
et on notera l'utilité de l'omniprésent `lambda`.

Sous-ensembles I

Les sous-ensembles de $\{a, b, c\}$ sont :

$$\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}.$$

On pourrait générer séparément les sous-ensembles de 0, de 1, de 2 et de 3 éléments mais l'usage direct d'un schéma récursif est plus simple.

Représentation d'un ensemble : liste sans répétition.

On examine comment la liste des sous-ensembles de $\{a, b, c\}$ se construit au départ de la liste des sous-ensembles de $\{b, c\}$, c'est-à-dire

$$\{\}, \{b\}, \{c\}, \{b, c\}.$$

On observe d'abord que les sous-ensembles de $\{b, c\}$ sont aussi des sous-ensembles de $\{a, b, c\}$, mais que la réciproque n'est pas vraie ; les sous-ensembles manquants sont

$$\{a\}, \{a, b\}, \{a, c\}, \{a, b, c\}.$$

On observe ensuite que les sous-ensembles nouveaux sont les anciens dans lesquels on a inséré l'élément nouveau a.

Sous-ensembles II

```
(define subsets ;; version inefficace !
  (lambda (e)
    (if (null? e)
        '(())
        (append (subsets (cdr e)) (insert-in-all (car e) (subsets (cdr e)))))))
```

La fonction auxiliaire `insert-in-all` prend comme arguments un objet `x` et une liste de listes `ll`. Elle renvoie une liste de listes, dont les éléments sont ceux de `ll` préfixés de `x`.

```
(define insert-in-all
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons x (car ll)) (insert-in-all x (cdr ll))))))
```

Cette "solution" est correcte mais très inefficace : l'appel `(subsets e)`, quand `e` n'est pas vide, provoque deux appels récursifs à `(subsets (cdr e))`.

Sous-ensembles III

Version efficace :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        ((lambda (le) (append le (insert-in-all (car e) le)))
         (subsets (cdr e))))))
```

Variante équivalente, plus lisible :

```
(define subsets
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (subsets (cdr e))))
          (append rec (insert-in-all (car e) rec))))))
```

Partitions I

La structure de l'ensemble des partitions d'un ensemble donné est peu apparente, mais cela n'empêche pas l'usage de la récursion. Il est naturel de considérer d'abord un exemple. Les cinq partitions de $\{a, b, c\}$ sont

$$\{\{a\}, \{b\}, \{c\}\},$$
$$\{\{a\}, \{b, c\}\}, \{\{b\}, \{a, c\}\}, \{\{c\}, \{a, b\}\},$$
$$\{\{a, b, c\}\}.$$

Les partitions de $\{b, c\}$ sont

$$\{\{b\}, \{c\}\},$$
$$\{\{b, c\}\}.$$

On observe qu'une partition de $\{a, b, c\}$ est obtenue au départ d'une partition de $\{b, c\}$ selon deux techniques :

1. En insérant le singleton $\{a\}$ comme partie supplémentaire ;
   $\{\{b\}, \{c\}\}$ donne $\{\{a\}, \{b\}, \{c\}\}$;
   $\{\{b, c\}\}$ donne $\{\{a\}, \{b, c\}\}$.
2. En insérant l'élément a dans une partie existante ;
   $\{\{b\}, \{c\}\}$ donne $\{\{a, b\}, \{c\}\}$ et $\{\{b\}, \{a, c\}\}$;
   $\{\{b, c\}\}$ donne $\{\{a, b, c\}\}$.

Partitions II

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (append (procede-1 (car e) (partitions (cdr e)))
                (procede-2 (car e) (partitions (cdr e)))))))
```

Attention à l'(in)efficacité !

```
(define partitions
  (lambda (e)
    (if (null? e)
        '(())
        (let ((rec (partitions (cdr e))))
          (append (procede-1 (car e) rec)
                  (procede-2 (car e) rec))))))
```

Partitions III

```
(define procede-1
  (lambda (x lp) (insert-in-all (list x) lp)))

(define insert-in-all
  (lambda (x le)
    (if (null? le)
        '()
        (cons (cons x (car le)) (insert-in-all x (cdr le))))))

(define procede-2
  (lambda (x lp) (append-map (lambda (p) (split x p)) lp)))

(define append-map
  (lambda (f l)
    (if (null? l)
        '()
        (append (f (car l)) (append-map f (cdr l))))))

(define append-map (lambda (f l) (apply append (map f l))))
```

Partitions IV

La fonction `split` réalise le procédé 2 proprement dit, pour une partition. Comme toujours lorsque l'on spécifie une fonction auxiliaire, il convient de le faire de la manière la plus générale possible. Le second argument ne sera donc pas nécessairement une partition, mais une quelconque liste de listes.

```
(split 'a '((b) (c)))      (((a b) (c)) ((b) (a c)))
(split 'a '((b c)))        (((a b c)))
(split '2 '((1 2) () (3))) (((2 1 2) () (3)) ((1 2) (2) (3)) ((1 2) () (2 3)
```

Construction de `split` : facile par la tactique habituelle ;
comment obtient-on `(split x ll)` à partir de `(split x (cdr ll))` ?
Un exemple est toujours éclairant :

```
(split '0 '(() (3)))       ;; (split x (cdr ll))
          (((0) (3)) (() (0 3)))

(split '0 '((1 2) () (3))) ;; (split x ll)
          (((0 1 2) () (3)) ((1 2) (0) (3)) ((1 2) () (0 3)))
```

Partitions V

Le premier élément du résultat est à créer de toutes pièces ; c'est [[`(cons (cons x (car ll)) (cdr ll))`]]. Le reste s'obtient en remplaçant dans [[`(split x (cdr ll))`]] (liste de listes de listes) chaque élément [[`ss`]] (liste de listes) par [[`(cons (car ll) ss)`]].

```
(define split
  (lambda (x ll)
    (if (null? ll)
        '()
        (cons (cons (cons x (car ll)) (cdr ll))
              (map (lambda (ss) (cons (car ll) ss))
                   (split x (cdr ll)))))))
```

On peut à présent utiliser la fonction `partition` :

```
(partitions '(a b c))  ==>
(((a) (b) (c)) ((a) (b c)) ((a b) (c)) ((b) (a c)) ((a b c)))
```

Réduire le cas d'une liste non vide `l` au cas de `(cdr l)` est l'essentiel du travail d'application du schéma de récursion, mais résoudre le cas de la liste vide est tout aussi important.

Inversion de fonction I

On résout ici le problème de l'inversion d'une fonction réelle de variables réelles. Inverser la fonction

$$f : \mathbb{R}^3 \to \mathbb{R} : (x_1, x_2, x_3) \mapsto f(x_1, x_2, x_3)$$

par rapport à x_2 consiste à construire une fonction

$$g : \mathbb{R}^3 \to \mathbb{R} : (x_1, u, x_3) \mapsto g(x_1, u, x_3)$$

telle que

$$g(x_1, f(x_1, x_2, x_3), x_3) = x_2 \quad \text{et} \quad f(x_1, g(x_1, u, x_3), x_3) = u\,,$$

ou encore

$$g(x_1, u, x_3) = x_2 \quad \text{et} \quad f(x_1, x_2, x_3) = u\,,$$

pour toutes valeurs adéquates de x_2 et u. Le problème est mathématiquement difficile, puisque l'inverse n'existe pas toujours, mais devient plus simple dans le cas où la fonction à inverser est continue et strictement monotone (croissante ou décroissante) par rapport à l'argument (ici x_2) sur lequel porte l'inversion.

Inversion de fonction II

On ramène le cas général (n arguments, $p$ième argument) au cas particulier d'un premier argument sur lequel porte l'inversion, et d'une liste d'autres arguments. Des opérateurs comme `in` et `out` font la conversion, ici dans le cas $n = 3$, $p = 2$:

```
(define in
  (lambda (f)
    (lambda (x l) (f (car l) x (cadr l)))))

(define out
  (lambda (h)
    (lambda (x1 x2 x3) (h x2 (list x1 x3)))))

(define f (lambda (a b c) (+ a (* b c))))

(f 34 56 23)                    1322
((in f) 56 '(34 23))            1322
((out (in f)) 34 56 23)         1322
```

Inversion de fonction III

Fonction $(x, \ell) \mapsto f(x, \ell)$, fonction inverse $(u, \ell) \mapsto g(u, \ell)$
Les deux fonctions sont croissantes en leur premier argument.

Approximations successives : soit $(x_0, x_1, \ldots) \longrightarrow g(u, \ell)$
Si $f(x_n, \ell) > u$, alors $x_n > g(u, \ell)$ et on choisit $x_{n+1} < x_n$;
si $u > f(x_n, \ell)$, on choisit $x_{n+1} > x_n$.

Deux variantes : $\mathbb{R} \to \mathbb{R}$ et $\mathbb{R}^+ \to \mathbb{R}^+$. On passe de l'une à l'autre par transformation exponentielle ou logarithmique. La seconde variante est développée ici.

Technique de la bissection. On maintient un intervalle $[m, M]$ dans lequel la valeur recherchée se trouve, on le rétrécit à chaque itération. Au départ, l'intervalle est très grand, par exemple $m = 10^{-6}$ et $M = 10^7$. A chaque étape, on calcule la moyenne (géométrique) μ des bornes de l'intervalle puis la valeur $f(\mu, \ell)$. En fonction de cette valeur, on décide de s'arrêter, si l'écart entre $f(\mu, \ell)$ et u n'excède pas une certaine quantité ε_y ou de continuer soit avec l'intervalle $[m, \mu]$, soit avec l'intervalle $[\mu, M]$. Chaque étape a pour effet de réduire la longueur de l'intervalle (de moitié, pour la première variante) et on peut s'arrêter dès que cette longueur devient moindre qu'une certaine quantité ε_x.

Inversion de fonction IV

Variables globales, fonctions auxiliaires :

```
(define *min* 1.e-6)            (define *eps-x* 1.e-12)
(define *max* 1.e+7)            (define *eps-y* 1.e-12)
(define mu (lambda (a b) (sqrt (* a b))))
(define prox (lambda (a b eps) (< (abs (- a b)) eps)))

(define inv+    ;; Cas où la fonction f est croissante
  (lambda (f)
    (lambda (u l)
      (i+ f u l *min* *max* *eps-x* *eps-y*))))

(define i+
  (lambda (f u l x0 x1 epsx epsy)
    (cond ((prox x0 x1 epsx) (mu x0 x1))
          ((prox (f (mu x0 x1) l) u epsy) (mu x0 x1))
          ((< (f (mu x0 x1) l) u) (i+ f u l (mu x0 x1) x1 epsx epsy))
          (else (i+ f u l x0 (mu x0 x1) epsx epsy)))))
```

Inversion de fonction V

L'opérateur `inv+` prend comme argument une fonction `f` croissante en son premier argument et renvoie la fonction inverse.

Le prédicat `prox` prend comme arguments deux réels a et b et un réel positif ε ; il renvoie vrai si $|a - b| < \varepsilon$.

Fonction `i+`

> *Si la fonction* $f : (\mathbb{R} \times \mathbb{R}^k) \to \mathbb{R}$ *est croissante en son premier argument et si l'unique solution* x *de l'équation* $f(x, \ell) = u$ *appartient à l'intervalle* $[x_0 : x_1]$*, alors* $i^+(f, u, \ell, x_0, x_1, \varepsilon_x, \varepsilon_y)$ *est un nombre* x' *proche de* x*, au sens que* x' *ne s'écarte pas de* x *de plus de* ε_x *ou que* $f(x', \ell)$ *ne s'écarte pas de* u *de plus de* ε_y*.*

On définit de manière analogue un opérateur `inv-` pour inverser les fonctions décroissantes, qui fera appel à l'opérateur auxiliaire `i-` ; ce dernier ne diffère de `i+` que par la permutation des comparateurs < et > dans les conditions des clauses contenant les appels récursifs.

Inversion de fonction VI

Un dernier point intéressant consiste à modifier `i+` de manière à éviter l'évaluation multiple des expressions `(mu x0 x1)` et `(f (mu x0 x1) l)`.

```
(define i+        ;; avec évaluation multiple
  (lambda (f u l x0 x1 epsx epsy)
    ((lambda (aux1)
       (cond ((prox x0 x1 epsx) aux1)
             ((prox (f aux1 l) u epsy) aux1)
             ((< (f aux1 l) u) (i+ f u l aux1 x1 epsx epsy))
             (else (i+ f u l x0 aux1 epsx epsy))))
     (mu x0 x1))))

(define i+        ;; sans évaluation multiple de aux1
  (lambda (f u l x0 x1 epsx epsy)
    (let ((aux1 (mu x0 x1)))
       (cond ((prox x0 x1 epsx) aux1)
             ((prox (f aux1 l) u epsy) aux1)
             ((< (f aux1 l) u) (i+ f u l aux1 x1 epsx epsy))
             (else (i+ f u l x0 aux1 epsx epsy))))))
```

Inversion de fonction VII

On peut réutiliser la même technique pour éviter la double évaluation de l'expression `(f aux1 l)` ; on obtient ainsi, sans utiliser `let` :

```
(define i+        ;; sans évaluation multiple de aux1 et aux2
  (lambda (f u l x0 x1 epsx epsy)
    ((lambda (aux1)
       ((lambda (aux2)
          (cond ((prox x0 x1 epsx) aux1)
                ((prox aux2 u epsy) aux1)
                ((< aux2 u) (i+ f u l aux1 x1 epsx epsy))
                (else (i+ f u l x0 aux1 epsx epsy))))
        (f aux1 l)))
     (mu x0 x1))))
```

Inversion de fonction VIII

Version utilisant `let*` :

```
(define i+        ;; sans évaluation multiple de aux1 et aux2
  (lambda (f u l x0 x1 epsx epsy)
    (let* ((aux1 (mu x0 x1))
           (aux2 (f aux1 l)))
      (cond ((prox x0 x1 epsx) aux1)
            ((prox aux2 u epsy) aux1)
            ((< aux2 u) (i+ f u l aux1 x1 epsx epsy))
            (else (i+ f u l x0 aux1 epsx epsy))))))
```

Inversion de fonction IX

$$f_1 : (x, [a, b]) \mapsto x(a + x(b + x))$$

$$f_1 : (x, [a, b]) \mapsto ax + bx^2 + x^3$$

```
(define (f1 x l) (* x (+ (car l) (* x (+ (cadr l) x)))))

(map (lambda (x) (f1 x '(1 1))) '(0 1 2 3 4 5 6))    (0 3 14 39 84 155 258)

(map (lambda (u) ((inv+ f1) u '(1 1))) '(0 3 14 39 84 155 258))
     (1.0000004460455906e-6 ;; précision médiocre
      1.0000000000001634    ;; précision excellente
      2.000000000000033     ;; précision excellente
      3.000000000000123     ;; précision excellente
      3.999999999999036     ;; précision excellente
      5.000000000000184     ;; précision excellente
      6.000000000000003)    ;; précision excellente
```

Problème du prêt I

Je souhaite emprunter de l'argent, pour acheter une maison et une voiture. J'ai contacté divers organismes prêteurs qui m'ont proposé différentes combinaisons de délais et de taux ; d'autres n'annoncent pas de taux mais directement le montant de la mensualité. Dans les rares cas où le taux et la mensualité étaient annoncés, j'ai recalculé la mensualité moi-même . . . et abouti à un montant inférieur à celui exigé. Curieusement, la différence tend à être plus importante pour les taux "voiture" que pour les taux "maison". D'où viennent ces divergences, variables d'une banque à l'autre mais systématiquement en ma défaveur ? Comment puis-je vérifier, et comparer différentes propositions ?

Problème du prêt II

L'emprunteur reçoit du prêteur une somme S et s'engage à la rembourser, accrue des intérêts, sous forme de "périodicités" constantes. On fixe un taux t et un nombre de périodes n. Supposons pour fixer les idées que la période est le mois et que le taux fixé est mensuel. On observe d'abord qu'une somme S, au terme d'un nombre n de mois, vaudra

$$S' = S(1+t)^n ; \qquad \text{(ic1)}$$

c'est la formule classique des intérêts composés. Dans le cas de remboursements mensuels constants, la formule devient

$$S' = M(1+t)^{n-1} + M(1+t)^{n-2} + \cdots + M(1+t) + M ; \qquad \text{(ic2)}$$

le $i$ème terme $M(1+t)^{n-i}$ représente la valeur à l'échéance (au terme du $n$ième mois) de la mensualité M payée au terme du $i$ème mois, et qui s'est donc valorisée pendant $(n-i)$ mois. On utilise la formule bien connue

$$\sum_{i=0}^{n-1} b^i = \frac{b^n - 1}{b-1} ,$$

où $b = 1+t$, pour effectuer la somme des valorisations des n mensualités et, par élimination de S' entre ic1 et ic2, on tire

$$S(1+t)^n = M\frac{(1+t)^n - 1}{t} \qquad \text{(ic3)}$$

ou encore

$$M = \frac{St(1+t)^n}{(1+t)^n - 1} \qquad \text{(ic4)}$$

Problème du prêt III

```
(define periodicite
  (lambda (S t n)
    (/ (* S t (expt (+ 1 t) n)) (- (expt (+ 1 t) n) 1))))

(define periodicite
  (lambda (S t n)
    (let ((aux (expt (+ 1 t) n))) (/ (* S t aux) (- aux 1)))))
```

La fonction `periodicite` permet de calculer la mensualité en fonction de la somme à emprunter, du taux d'intérêt mensuel et de la durée du prêt en mois.
En pratique, l'emprunteur qui connaît la mensualité requise, ou sa capacité maximale de remboursement, peut se poser les trois questions suivantes :

Etant donné que ma capacité de remboursement mensuel est de M euros,

- quelle somme maximale puis-je emprunter au taux mensuel t, pour n mois ?
- à quel taux mensuel maximal t puis-je emprunter la somme S, remboursable en n mois ?
- en combien de mois minimum puis-je rembourser la somme S empruntée au taux mensuel t ?

Problème du prêt IV

Les trois programmes d'inversion sont construits de la même manière :

```
(define periodicite<-somme
  (lambda (S tn) (periodicite S (car tn) (cadr tn))))

(define somme<-periodicite (inv+ periodicite<-somme))

(define somme
  (lambda (M t n) (somme<-periodicite M (list t n))))

(periodicite 100000 0.004 240)          648.957469845475
(somme 648.957469845475 0.004 240)      100000.0
```

Problème du prêt V

```
(define periodicite<-taux
  (lambda (t Sn) (periodicite (car Sn) t (cadr Sn))))
(define taux<-periodicite (inv+ periodicite<-taux))
(define taux (lambda (S M n) (taux<-periodicite M (list S n))))

(define periodicite<-nombre-periodes
  (lambda (n St) (periodicite (car St) (cadr St) n)))
(define nombre-periodes<-periodicite
  (inv- periodicite<-nombre-periodes))
(define nombre-periodes
  (lambda (S t M) (nombre-periodes<-periodicite M (list S t))))

;; (periodicite 100000 0.004 240)                    648.957469845475
;; (somme 648.957469845475 0.004 240)                100000.0
;; (taux 100000 648.957469845475 240)                0.0039999999997
;; (nombre-periodes 100000 0.004 648.957469845475)   240.000000000001
```

Problème du prêt VI

Première source de divergence : “simplifier” la conversion entre données annuelles et données mensuelles. Deux variantes existaient (et ont récemment été rendues illégales) :

— Considérer que le taux mensuel à appliquer est le douzième du taux annuel affiché ;
— Considérer que la mensualité à payer est le douzième de l'annuité calculée sur base du taux annuel affiché.

La première méthode est systématiquement défavorable à l'emprunteur car

$$(1 + t_a) = (1 + t_m)^{12} > 1 + 12t_m \,.$$

Les passages du taux annuel au taux mensuel (exact) et réciproquement se programment aisément :

```
(define tm<-ta
  (lambda (ta) (- (expt (+ ta 1) 1/12) 1)))

(define ta<-tm
  (lambda (tm) (- (expt (+ tm 1) 12) 1)))
```

L'organisme prêteur qui applique un taux mensuel effectif égal au douzième de son taux annuel nominal utilise en fait un taux annuel effectif supérieur au taux nominal annoncé ; pour savoir quel taux annuel lui sera réellement appliqué, le client pourra utiliser le programme suivant :

```
(define ta<-ta_1 (lambda (t) (ta<-tm (/ t 12))))
; exemple:  (ta<-ta_1 0.06)   .0616778
```

Problème du prêt VIII

La deuxième variante est, elle aussi, défavorable à l'emprunteur, qui rembourse chaque mensualité avec une avance de 1 à 11 mois, c'est-à-dire avec, en moyenne, cinq mois et demi d'avance. Ici, le désavantage dépend non seulement du taux mais aussi de la durée du prêt. On utilise le programme

```
(define compose
  (lambda (f g) (lambda (x) (f (g x)))))

(define compose-list
  (lambda (f-list)
    (if (null? f-list)
        (lambda (x) x)
        (compose (car f-list)
                 (compose-list (cdr f-list))))))
```

Problème du prêt IX

```
(define ta<-ta-2
  (compose-list
    (list ta<-tm tm<-mensu mensu<-annu annu<-ta-2)))

(define annu<-ta-2
  (lambda (t) (periodicite *S* t *n*)))

(define mensu<-annu
  (lambda (a) (/ a 12.0)))

(define tm<-mensu
  (lambda (M) (taux *S* M (* 12 *n*))))

(define ta<-tm
  (lambda (tm) (- (expt (+ tm 1) 12) 1)))
```

C'est pire que précédemment, surtout pour les courtes durées :

```
; *n* 20 :  (ta<-ta-2 .06)   .063523
; *n* 10 :  (ta<-ta-2 .06)   .066294
; *n*  3 :  (ta<-ta-2 .06)   .079255
```

Problème du prêt X

Solution sans variables globales gênantes :

```
(define make-ta<-ta-2
  (lambda (n)
    (compose-list
 ; ta<-tm ; tm<-mensu ; mensu<-annu ; annu<-ta-2
      (list
        (lambda (tm) (- (expt (+ tm 1) 12) 1))
        (lambda (M) (taux *S* M (* 12 n)))
        (lambda (a) (/ a 12.0))
        (lambda (t) (periodicite *S* t n))))))
```

On notera que `make-ta<-ta-2` est une fonction qui à tout nombre positif (représentant la durée du prêt) associe une fonction équivalente à `ta<-ta-2` donnée plus haut.

Problème du prêt XI

En reprenant les mêmes exemples que précédemment, on obtient

```
(define *S* 1234567)   ... ;; S quelconque
((make-ta<-ta-2 20) .06)   .063523
((make-ta<-ta-2 10) .06)   .066294
((make-ta<-ta-2  3) .06)   .079255
```

La technique consistant à écrire un “générateur de fonction” permet d’éviter les variables globales gênantes sans que l’on doive modifier le nombre d’arguments des fonctions impliquées.

Problème du prêt XII

Deuxième cause de divergence : la notion de taux d'intérêt est parfois remplacée par la notion "voisine" de taux de chargement. L'emprunteur d'une somme S remboursable en n mois au taux de chargement mensuel t_c rembourse chaque mois la somme $M = S'/n = (1/n + t_c)S$.

Calcul de la mensualité à partir de la somme à emprunter, du taux de chargement ou du taux d'intérêt et de la durée du prêt en mois :

```
(define mensualite<-tm periodicite)

(define tm<-mensualite taux)

(define mensualite<-tc
  (lambda (S tc n) (* (+ (/ 1 n) tc) S)))

(define tc<-mensualite
  (lambda (S M n) (- (/ M S) (/ 1 n))))
```

On en tire immédiatement les fonctions de conversion :

```
(define tc<-tm
  (lambda (tm n) (tc<-mensualite *S* (mensualite<-tm *S* tm n) n)))

(define tm<-tc
  (lambda (tc n) (tm<-mensualite *S* (mensualite<-tc *S* tc n) n)))
```

Problème du prêt XIII

On calcule d'abord quel taux d'intérêt mensuel correspond à un taux de chargement de 0.5 %, pour des durées de prêt de 1, 12, 30 et 240 mois ; on obtient

```
(tm<-tc 0.005   1)   0.005000
(tm<-tc 0.005  12)   0.009080
(tm<-tc 0.005  30)   0.009265
(tm<-tc 0.005 240)   0.007719
```

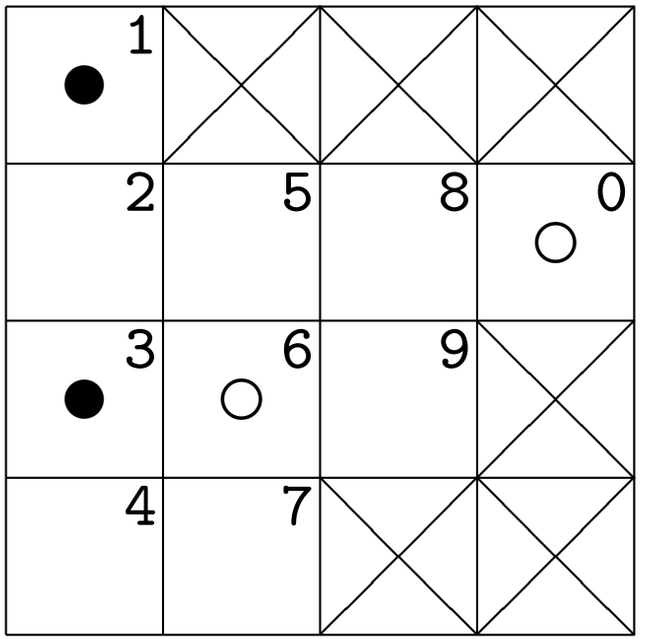
On calcule de même quel taux de chargement correspond à un taux d'intérêt mensuel de 0.5 % :

```
(tc<-tm 0.005   1)   0.005000
(tc<-tm 0.005  12)   0.002733
(tc<-tm 0.005  30)   0.002646
(tc<-tm 0.005 240)   0.002998
```

On voit que la confusion entre les deux notions de taux peut coûter cher !

Problème des Cavaliers I

Comment permuter les Cavaliers,
les cases barrées étant interdites ?



Configuration `(a b c d)` :

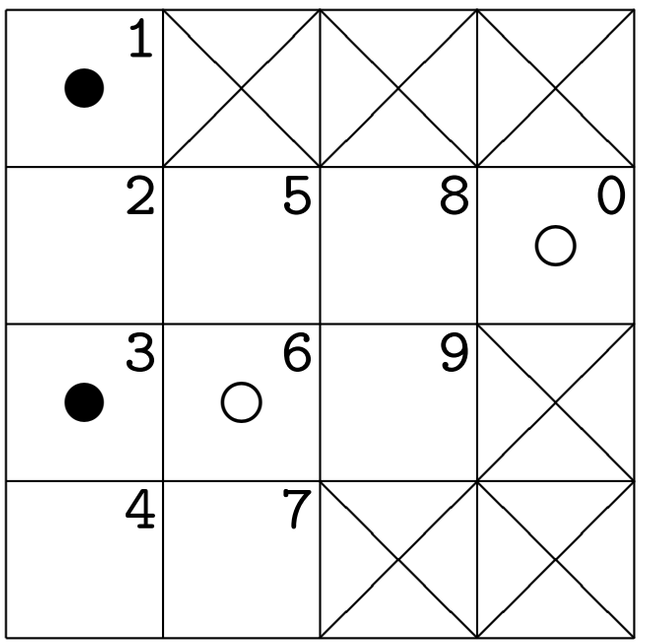
le Cavalier noir se trouvant initialement en `1` se trouve en `a` ;
le Cavalier noir se trouvant initialement en `3` se trouve en `b` ;
le Cavalier blanc se trouvant initialement en `6` se trouve en `c` ;
le Cavalier blanc se trouvant initialement en `0` se trouve en `d`.

Mouvement `(a . b)` : de la position `a` vers la position `b`.

Problème des Cavaliers II

Variables globales : situation initiale, liste des mouvements possibles :

```
(define *init* '(1 3 6 0))


(define *moves*
  '((1 . 6) (1 . 8) (2 . 7) (2 . 9)
    (3 . 8) (4 . 5) (4 . 9) (5 . 4)
    (6 . 1) (6 . 0) (7 . 2) (7 . 8)
    (8 . 1) (8 . 3) (8 . 7) (9 . 2)
    (9 . 4) (0 . 6)))
```



On voit par exemple qu'un Cavalier se trouvant en position 8 peut aller en un coup à la position 1, 3 ou 7 (si cette position est libre).

Problème des Cavaliers III

Les situations finales acceptables sont peu nombreuses ; en effet, la liste `(1 3 6 0)` admet $4! = 24$ permutations, dont 9 sont des dérangements (permutations sans point fixe).

```
(define *final* '((0 1 3 6) (0 6 1 3) (0 6 3 1)
                  (3 0 1 6) (3 1 0 6) (3 6 0 1)
                  (6 0 1 3) (6 0 3 1) (6 1 0 3)))
```

Si on se limite aux permutations dans lesquelles les Cavaliers blancs prennent la place des Cavaliers noirs (et réciproquement), on a :

```
(define *best* '((0 6 1 3) (0 6 3 1) (6 0 1 3) (6 0 3 1)))
```

```
(define n= (lambda (x y) (not (= x y))))  ;; case libre?
(define k1 car)   (define k2 cadr)  (define k3 caddr)  (define k4 cadddr)
(define org car)  (define dst cdr)        ;; accesseurs
```

Problème des Cavaliers IV

La fonction centrale du problème est la fonction `succ`. Elle prend comme arguments une situation et un mouvement ; si ce mouvement est possible dans la situation donnée, la fonction renvoie la situation résultante, sinon elle renvoie `#f`. On a :

```
(define succ
  (lambda (sit mv)
    (let ((a (k1 sit)) (b (k2 sit)) (c (k3 sit)) (d (k4 sit))
          (e (org mv)) (f (dst mv)))
      (cond
        ((and (= a e) (n= b f) (n= c f) (n= d f)) (list f b c d))
        ((and (= b e) (n= a f) (n= c f) (n= d f)) (list a f c d))
        ((and (= c e) (n= a f) (n= b f) (n= d f)) (list a b f d))
        ((and (= d e) (n= a f) (n= b f) (n= c f)) (list a b c f))
        (else #f)))))
```

Les cinq clauses de la forme `cond` correspondent aux cinq cas possibles : l'origine du mouvement correspond à l'une des quatre positions de la situation donnée ou à aucune.

Problème des Cavaliers V

Sur base de la fonction `succ`, on peut définir une fonction `succs` prenant comme arguments une situation et une liste de mouvements et renvoyant la liste triée (ordre lexicographique) et sans répétitions des situations résultantes.

```
(define succs
  (lambda (sit mvs)
    (if (null? mvs)
        '()
        (let ((s1 (succ sit (car mvs))) (s (succs sit (cdr mvs))))
          (if s1 (insert-sit s1 s) s)))))
(define insert-sit (lambda (sit sits) (add-elem sit sits (lex < =))))
(succs *init* *moves*) ((1 8 6 0) (8 3 6 0))
```

Une situation σ appartient à [[`(succs sit mvs)`]] s'il existe un mouvement μ appartenant à [[`mvs`]] qui permette de passer de [[`sit`]] à σ.
La condition d'un `if` est assimilée à vrai dès qu'elle n'est pas fausse ; c'est pourquoi il était intéressant que la fonction précédente `succ` renvoie `#f` pour signaler l'absence de situation résultante.

Problème des Cavaliers VI

```
(define succss
  (lambda (sits mvs)
    (if (null? sits)
        '()
        (let ((l1 (succs (car sits) mvs)) (l (succss (cdr sits) mvs)))
          (merge-sits l1 l)))))

(define merge-sits          ;; fusion de
  (lambda (s1 s2)           ;; deux listes triées
    (cond ((null? s1) s2)
          ((null? s2) s1)
          ((equal? (car s1) (car s2))
           (cons (car s1) (merge-sits (cdr s1) (cdr s2))))
          (((lex < =) (car s1) (car s2))
           (cons (car s1) (merge-sits (cdr s1) s2)))
          (else (cons (car s2) (merge-sits s1 (cdr s2)))))))
```

Une situation σ appartient à [[`(succs sits mvs)`]] s'il existe une situation ρ appartenant à [[`sits`]] et un mouvement μ appartenant à [[`mvs`]] qui permette de passer de ρ à σ.

Problème des Cavaliers VII

La fonction `succss` permet en principe de résoudre le problème posé.
En effet, par applications successives, on peut construire l'ensemble des situations accessibles depuis la situation initiale en un coup, en deux coups, etc. Cette approche naïve est peu efficace car elle ne tient pas compte des cycles et, comme tout mouvement est réversible, les cycles sont nombreux.

On définit donc une fonction `new-succss` qui ne donne que les "nouveaux" successeurs, c'est-à-dire les situations non encore rencontrées.
Les situations déjà rencontrées, et donc dorénavant interdites, sont groupées dans un troisième argument. On a :

```
(define new-succss
  (lambda (sits mvs forbid)
    (let ((ss (succss sits mvs)))
      (diff ss forbid))))          ;; diff: différence ensembliste
```

$\sigma \in$ [[`(new-succs sits mvs forbid)`]] si et seulement si
$\sigma \in$ [[`(succs sits mvs)`]] et $\sigma \notin$ [[`forbid`]].

Problème des Cavaliers VIII

```
(define gen                              ;; RESOUT LE PROBLEME
  (lambda (n inits finals)
    (if (= n 0)
        (list 0 '() inits 0 1 '())
        (let* ((rec1 (gen (- n 1) inits finals))
               (rec (cdr rec1)))
          (let ((old (car rec)) (new (cadr rec))
                (lold (caddr rec)) (lnew (cadddr rec)))
            (let ((old1 (merge old new))
                  (new1 (new-succss new *moves* old)))
              (let ((lold1 (+ lold lnew))
                    (lnew1 (length new1))
                    (int (inter new1 finals)))
                (newline)
                (display (list n lold1 lnew1 int))
                (list n old1 new1 lold1 lnew1 int)))))))))
```

Résultats :

`n`	nombre de coups
`old1`	situations accessibles en moins de `n` coups
`new1`	situations accessibles en `n` coups mais pas moins
`lold1`	longueur de `old1`
`lnew1`	longueur de `new1`
`int`	situations finales accessibles en `n` coups mais pas moins

Problème des Cavaliers IX

La fonction `gen` imprime des résultats intermédiaires intéressants.

```
(gen 4 (list *init*) *final*)
    (1 1 2 ())   ;; premier coup, deux nouvelles situations
    (2 3 3 ())   ;; deuxième coup, trois nouvelles situations
    (3 6 6 ())   ;; troisième coup, six nouvelles situations
    (4 12 11 ()) ;; quatrième coup, onze nouvelles situations
```

```
(4
 ((1 2 6 0) (1 3 6 0) (1 7 6 0) (1 8 6 0) (2 3 6 0) (7 3 1 0)
  (7 3 6 0) (7 8 6 0) (8 3 1 0) (8 3 1 6) (8 3 6 0) (8 7 6 0))
 ((1 9 6 0) (2 3 1 0) (2 8 6 0) (3 7 6 0) (7 1 6 0) (7 3 1 6)
  (7 3 8 0) (7 8 1 0) (8 2 6 0) (8 7 1 0) (9 3 6 0))
 12   ;; 12 situations accessibles en moins de quatre coups.
 11   ;; 11 situations accessibles en quatre coups mais pas moins,
 ())  ;;    dont aucune n'est finale.
```

Problème des Cavaliers X

La valeur renvoyée est une liste de six résultats. Le premier rappelle que quatre coups consécutifs ont été joués. Le second résultat et le quatrième donnent la liste des situations accessibles en moins de quatre coups et leur nombre (12). Le troisième résultat et le cinquième donnent la liste des situations accessibles en quatre coups mais pas moins, et leur nombre (11). Le sixième résultat est la liste des situations finales accessibles en quatre coups mais pas moins ; on constate que cette liste est vide. De plus, en cours d'exécution, des résultats s'affichent pour un, deux, trois et quatre coups ; ces résultats sont élagués : ils comprennent les longueurs des listes de situations générées, mais pas ces listes elles-mêmes, sauf pour la liste des situations finales accessibles.

Remarque. Il faut éviter la construction d'objets trop gros, et pour cela évaluer *a priori* la taille des objets construits. Dans le cas présent, la taille maximale d'une liste de situations est le nombre de quadruplets ordonnés de nombres distincts compris entre 0 et 9 ; ce nombre est $10 * 9 * 8 * 7 = 5\,040$.

Problème des Cavaliers XI

```
(gen 49 (list *init*) *final*)
    (1 1 2 ())
    ...
    (25 2035 191 ())
    (26 2226 171 ((3 0 1 6) (6 1 0 3)))
    (27 2397 145 ())
    (28 2542 128 ((0 1 3 6) (3 1 0 6) (3 6 0 1)))
    (29 2670 120 ())
    ...
    (39 4166 181 ())
    (40 4347 173 ((0 6 1 3) (6 0 1 3) (6 0 3 1)))
    (41 4520 152 ())
    (42 4672 120 ((0 6 3 1)))
    (43 4792 94 ())
    ...
    (49 5037 3 ())
(49  ;; DETERMINE EXPERIMENTALEMENT
 ((0 1 2 3) (0 1 2 4) ... (9 8 7 5) (9 8 7 6))
 ((2 4 9 5) (2 9 4 5) (9 2 4 5))
 5037
 3
 ())
```

Problème des Cavaliers XII

Conclusions :

- Le nombre de coups conduisant à une situation finale est de 26 au minimum.
- Le nombre de coups conduisant à une situation finale permutant les couleurs est de 40 au minimum.
- Toute situation est accessible en moins de 50 coups.
- Les trois situations les moins accessibles sont `(2 4 9 5)`, `(2 9 4 5)` et `(9 2 4 5)` ; 49 coups sont nécessaires.

Le problème des Cavaliers n'est que l'un des nombreux représentants de la classe des problèmes dits "de recherche" ou "d'exploration dans un espace d'état". Les problèmes de cette classe comportent tous un certain ensemble structuré (ici, les 5 040 situations possibles) et un chemin à déterminer dans cette ensemble (ici, une suite de mouvements). Fondamentalement, il n'y a aucune difficulté si ce n'est la taille de l'espace et le nombre potentiellement très élevé de chemins possibles.

Problème des Cavaliers XIII

Dans le cas présent, aucune tactique n'est requise pour réduire le travail de recherche, car 5 040 situations correspondent à un espace très réduit. Néanmoins, à titre d'illustration, nous présentons une tactique dont l'emploi est fréquemment nécessaire en pratique.

La fonction `gen` construit les chemins depuis la situation initiale jusqu'à la situation finale. Faire l'inverse ne serait ni plus ni moins efficace puisque chaque mouvement est réversible.

Ce n'est pas le cas pour tous les problèmes de recherche et, parfois, enchaîner les mouvements vers l'arrière plutôt que vers l'avant accroît très significativement l'efficacité de la recherche.

On peut attendre une amélioration de l'efficacité en combinant une recherche vers l'avant et une recherche vers l'arrière. La variante `gen2` construit les chemins à partir de leurs deux extrémités, de manière symétrique.

Problème des Cavaliers XIV

```
(define (gen2 n inits finals)
  (if (= n 0)
      (list 0 (list '() inits 0 1) (list '() finals 0 9) '() '())
      (let* ((rec1 (gen2 (- n 1) inits finals))
             (recf (cadr rec1)) (recb (caddr rec1)))
        (let ((fold (car recf)) (fnew (cadr recf))
              (flold (caddr recf)) (flnew (cadddr recf))
              (bold (car recb)) (bnew (cadr recb))
              (blold (caddr recb)) (blnew (cadddr recb)))
          (let ((fold1 (merge fold fnew))
                (fnew1 (new-succss fnew *moves* fold))
                (bold1 (merge bold bnew))
                (bnew1 (new-succss bnew *moves* bold)))
            (let ((flold1 (+ flold flnew)) (flnew1 (length fnew1))
                  (blold1 (+ blold blnew)) (blnew1 (length bnew1))
                  (i-a (inter fold1 bnew1)) (i-b (inter fnew1 bnew1)))
              (newline)
              (display (list n i-a i-b))
              (list n (list fold1 fnew1 flold1 flnew1)
                      (list bold1 bnew1 blold1 blnew1) i-a i-b)))))))
```

Problème des Cavaliers XV

La liste renvoyée comporte le nombre n = [[n]] d'itérations, un quadruplet concernant la progression vers l'avant ("f" signifie "forward"), un quadruplet analogue concernant la progression vers l'arrière ("b" signifie "backward") et deux listes [[i-a]] et [[i-b]] comportant les situations apparaissant à l'intersection des fronts avant et arrière. Le quadruplet "avant" comporte les listes des situations accessibles par l'avant en moins de n coups et en exactement n coups, et les longueurs de ces listes. La liste [[i-a]] contient les situations accessibles par l'avant en moins de n pas et, simultanément, accessibles par l'arrière en exactement n pas. De même, la liste [[i-b]] contient les situations simultanément accessibles par l'avant et par l'arrière, en exactement n pas dans les deux cas. Ces deux listes, si elles ne sont pas vides, témoignent de l'existence d'un ou plusieurs chemin(s) de longueur moindre que $2n$, ou égale à $2n$, respectivement.

Problème des Cavaliers XVI

Le nombre d'itérations passe de 49 à 25.

```
(gen2 25 (list *init*) *final*)
    (1 () ())
    ...
    (12 () ())
    (13 () ((2 9 7 8) (9 8 2 7)))
    (14 ((2 9 7 1) (9 3 2 7)) ((2 4 7 8) ... (9 7 2 3)))
    ...
    (25 ((1 6 8 0) ... (8 6 3 0)) ())


(25
 (((0 2 1 3) (0 2 3 1) ... (9 8 7 6))
  ((0 2 1 6) ... (9 8 4 0))
  2035
  191)
 (((0 1 2 3) ...  (9 8 7 6))
  ((1 6 8 0) ...  (8 6 3 0))
  5001
  24)
 ((1 6 8 0) ... (8 6 3 0))
 ())
```

Problème des Cavaliers XVII

On a vu précédemment que 26 étapes suffisaient pour passer de la situation initiale `(1 3 6 0)` à l'une des situations finales `(3 0 1 6)` et `(6 1 0 3)` ; la présente exécution montre que les chemins réalisant ce transfert ont nécessairement pour étape médiane la situation `(2 9 7 8)` ou la situation `(9 8 2 7)`.

Remarques. Dans le cas présent, aborder le problème "des deux côtés", c'est-à-dire depuis la situation initiale et depuis la situation finale, n'abrège pas la recherche, puisque tout l'espace est exploré. Notons déjà que dans beaucoup de problèmes analogues, l'espace à explorer est trop grand pour être construit en entier, ce qui peut rendre nécessaire l'emploi d'une tactique plus fine que la recherche exhaustive et systématique. Un défaut potentiel des programmes présentés ici est qu'ils ne fournissent pas la suite de mouvements permettant de passer de la situation initiale à l'une des situations finales. On peut facilement adapter les programmes pour remédier à cette lacune, mais au prix d'une consommation de ressources nettement plus élevée. La raison en est que dans une situation donnée, plusieurs mouvements sont en général possibles, ce qui donne lieu à une explosion combinatoire du nombre de chemins à construire et à explorer. Parfois, cette explosion combinatoire peut être évitée par un raisonnement simple.

Problème des cruches I

Ce problème classique montre que, dans certains cas, l'explosion combinatoire peut être entièrement évitée. Plus précisément, une situation intermédiaire semble avoir plusieurs successeurs possibles, mais un seul d'entre eux mérite d'être considéré.

> *On dispose de deux cruches dont les contenances en litres sont respectivement a et b et d'une source inépuisable d'eau. On demande de prélever exactement c litres d'eau en un nombre minimum d'opérations. Les nombres a, b et c sont des entiers naturels distincts tels que $a < b$ et $c < a + b$. On admet qu'une cruche ne permet de prélever avec précision que sa contenance nominale.*

Il existe six manipulations possibles :

1. Remplir la petite cruche.
2. Remplir la grande cruche.
3. Transvaser de la petite cruche vers la grande.
4. Transvaser de la grande cruche vers la petite.
5. Vider la petite cruche.
6. Vider la grande cruche.

Problème des cruches II

Après remplissage, une cruche contient exactement sa contenance nominale. Transvaser une cruche dans l'autre ne modifie pas le contenu global des deux cruches. Un point important est qu'avant et après chaque opération, au plus une des deux cruches peut avoir un contenu autre que nul ou maximal, car toute opération, y compris le transvasement d'une cruche dans l'autre, a pour effet de remplir ou de vider une cruche.

Il est clair qu'à la première étape on a deux possibilités : remplir la petite cruche, ou remplir la grande. On observe aussi qu'à chaque étape ultérieure on n'a qu'une seule possibilité intelligente. En effet, exactement une étape sur deux est un transvasement (en provenance de la cruche pleine, s'il y en a une, ou à destination de la cruche vide sinon) ; ce transvasement a pour effet, soit de vider la cruche de départ, soit de remplir la cruche d'arrivée. Dans le premier cas, l'étape suivante consiste à remplir la cruche vide ; dans le second cas, l'étape suivante consiste à vider la cruche pleine.

Problème des cruches III

On représente une situation par une paire pointée dont les composants sont les contenus de la petite et de la grande cruche, respectivement.

Une situation est dite *initiale* si l'une des cruches est pleine et l'autre vide.

Une situation est dite *intéressante* si l'une des cruches est pleine ou vide tandis que l'autre n'est ni pleine ni vide.

Une situation est *convenable* si elle est initiale ou intéressante.

On définit une fonction **`next-sit`** à quatre arguments : les contenances **`p`** et **`g`** de la petite cruche et de la grande cruche, une situation intéressante **`sit`**, et l'un des symboles **`trans`** et **`in/out`**.

Cette fonction renvoie la seule situation convenable atteignable en un transvasement (si le quatrième argument est **`trans`**) ou en remplissant la cruche vide ou en vidant la cruche pleine (si le quatrième argument est **`in/out`**).

Problème des cruches IV

```
(define next-sit
  (lambda (p g sit op)
    (let ((p1 (car sit)) (g1 (cdr sit)))
      (if (eq? op 'trans)
          (cond ((= p1 p)
                 (let ((dg (- g g1)))
                   (if (> p dg)
                       (cons (- p dg) g)
                       (cons 0 (+ g1 p)))))
                ((= p1 0)
                 (if (> p g1)
                     (cons g1 0)
                     (cons p (- g1 p))))
                ((= g1 g) (cons p (- g1 (- p p1))))
                ((= g1 0) (cons 0 p1)))
          (cond ((= p1 p) (cons 0 g1))
                ((= p1 0) (cons p g1))
                ((= g1 g) (cons p1 0))
                ((= g1 0) (cons p1 g)))))))
```

Problème des cruches V

```
(define seq-sits
  (lambda (p g sit op past)
    (if (member sit past)
        past
        (let ((new-sit
               (next-sit p g sit op))
              (op1
               (if (eq? op 'trans)
                   'in/out
                   'trans)))
          (seq-sits p
                    g
                    new-sit
                    op1
                    (cons sit past))))))
```

[[`(seq-sits p g sit op past)`]] est la liste des situations que l'on peut atteindre au départ de [[`sit`]] en alternant `trans` et `in/out` et sans passer par une situation élément de [[`past`]] ; la première opération effectuée est [[`op`]] et les contenances des cruches sont [[`p`]] et [[`g`]].

Problème des cruches VI

Exemple : cruches de huit et onze litres.
Situations initiales convenables : `(8 . 0)` et `(0 . 11)`.

La fonction `seq-sits` permet de construire toutes les situations que l'on peut obtenir à partir des situations initiales ; on peut utiliser `reverse` pour que ces situations soient énumérées dans l'ordre naturel :

```
(reverse (seq-sits 8 11 '(8 . 0) 'trans '()))      ==>
((8 . 0) (0 . 8) (8 . 8) (5 . 11) (5 . 0) (0 . 5) (8 . 5) (2 . 11)
 (2 . 0) (0 . 2) (8 . 2) (0 . 10) (8 . 10) (7 . 11) (7 . 0) (0 . 7)
 (8 . 7) (4 . 11) (4 . 0) (0 . 4) (8 . 4) (1 . 11) (1 . 0) (0 . 1)
 (8 . 1) (0 . 9) (8 . 9) (6 . 11) (6 . 0) (0 . 6) (8 . 6) (3 . 11)
 (3 . 0) (0 . 3) (8 . 3) (0 . 11) (8 . 11))

(reverse (seq-sits 8 11 '(0 . 11) 'trans '()))     ==>
((0 . 11) (8 . 3) (0 . 3) (3 . 0) (3 . 11) (8 . 6) (0 . 6) (6 . 0)
 (6 . 11) (8 . 9) (0 . 9) (8 . 1) (0 . 1) (1 . 0) (1 . 11) (8 . 4)
 (0 . 4) (4 . 0) (4 . 11) (8 . 7) (0 . 7) (7 . 0) (7 . 11) (8 . 10)
 (0 . 10) (8 . 2) (0 . 2) (2 . 0) (2 . 11) (8 . 5) (0 . 5) (5 . 0)
 (5 . 11) (8 . 8) (0 . 8) (8 . 0) (0 . 0))
```

Problème des cruches VII

Si on souhaite prélever quinze litres, les situations finales adéquates sont `(8 . 7)` et `(4 . 11)` ; on observe dans les listes ci-dessus que 17 étapes sont nécessaires, puisque la paire `(8 . 7)` est le 17ième élément de la première liste ; la paire `(4 . 11)` n'apparaît qu'en 19ième position, dans la seconde liste.

Abstraction faite du dernier élément de chacune d'elles, ces listes sont inverses l'une de l'autre. La longueur commune de ces listes est 36.

Deux situations séparées par un transvasement ne sont pas essentiellement différentes, en ce sens que le contenu total des deux cruches est le même ; il y a donc 18 situations essentiellement différentes, correspondant à tous les contenus totaux possibles, de 1 à 18 litres.

A posteriori, il est évident que, si les contenances des deux cruches sont a et b, il y a au maximum $a + b - 1$ situations intéressantes essentiellement différentes. On peut démontrer qu'il y en a exactement $a + b - 1$ si a et b sont premiers entre eux.

10. Abstraction sur les données

Motivation. Souvent, les données et résultats d'un même problème peuvent se représenter concrètement (en machine) de plusieurs manières, chacune pouvant avoir ses avantages. Changer de structures de données concrètes implique de nombreuses modifications éparses dans les programmes, sauf si le programmeur a "prévu le coup".

Exemple I. On représentera le plus souvent un rationnel par une paire d'entiers (*num*,*den*). On a deux possibilités : n'admettre que la forme réduite (dénominateur positif, numérateur et dénominateur premiers entre eux) ou autoriser aussi les formes non réduites. Dans le premier cas, il faut décider si la réduction a lieu dès que le rationnel est construit, ou seulement quand il est utilisé et/ou affiché.

Exemple II. On représentera le plus souvent un polynôme tel que $3x^5 + 2x^3 - 4x - 7$ par une liste :
`(3 0 2 0 -4 -7)` ou `((5 . 3) (3 . 2) (1 . -4) (0 . -7))`.

La première solution est préférable pour les polynômes "pleins", la seconde pour les polynômes "creux".

Conclusion. Mieux vaut laisser toutes les possibilités ouvertes, et minimiser et localiser au mieux les fragments de programmes dépendant de la représentation adoptée.

Principe

On peut manipuler les listes au moyen d'un constructeur `cons` et d'accesseurs `car` et `cdr`, sans savoir comment ces procédures (et les listes elles-mêmes) sont réalisées.

L'utilisateur peut aussi définir "axiomatiquement" des données abstraites, en fixant d'abord les primitives : constructeur(s) et accesseur(s) ; ces données sont alors réalisées en programmant les primitives.

On peut imaginer par exemple les *rationnels abstraits*, basés sur le constructeur `make-ratl` et les accesseurs `numr` et `denr`. On sait que, si n et d sont des entiers ($d \neq 0$), (`make-ratl` n d) est un rationnel égal à n/d ; d'autre part, si r est un rationnel, alors les valeurs de (`numr r`) et (`denr r`) sont les numérateur et dénominateur d'une fraction (réduite où non) correspondant à r.

On considère séparément les problèmes d'utilisation des rationnels (via le constructeur `make-ratl` et les accesseurs `numr` et `denr`) et le problème de la réalisation de ces derniers (en termes de primitives Scheme).

On peut faire de même pour les polynômes. Un polynôme est, soit le polynôme nul, soit la somme d'un monôme (degré et coefficient) et d'un polynôme de degré inférieur.

Manipulation de rationnels abstraits I

Ces programmes seront valables, que l'on travaille avec des fractions réduites ou non, et quel que soit le mode de représentation d'une fraction.

```
(define rzero?
  (lambda (rtl) (zero? (numr rtl))))

(define r+
  (lambda (x y)
    (make-ratl (+ (* (numr x) (denr y)) (* (numr y) (denr x)))
               (* (denr x) (denr y)))))

(define r*
  (lambda (x y)
    (make-ratl (* (numr x) (numr y))
               (* (denr x) (denr y)))))

(define r-
  (lambda (x y)
    (make-ratl (- (* (numr x) (denr y)) (* (numr y) (denr x)))
               (* (denr x) (denr y)))))
```

Manipulation de rationnels abstraits II

```
(define rinvert
  (lambda (rtl)
    (if (rzero? rtl)
        (error "rinvert: Cannot invert " rtl)
        (make-ratl (denr rtl) (numr rtl)))))

(define r/ (lambda (x y) (r* x (rinvert y))))

(define r=
  (lambda (x y) (= (* (numr x) (denr y)) (* (numr y) (denr x)))))

(define rpositive?
  (lambda (rtl)
    (or (and (positive? (numr rtl)) (positive? (denr rtl)))
        (and (negative? (numr rtl)) (negative? (denr rtl))))))

(define r> (lambda (x y) (rpositive? (r- x y))))
```

Manipulation de rationnels abstraits III

```
(define max
  (lambda (x y)
    (if (> x y) x y)))

(define rmax
  (lambda (x y)
    (if (r> x y) x y)))

(define extreme-value
  (lambda (pred x y)
    (if (pred x y) x y)))

(define rprint
  (lambda (rtl)
    (writeln (numr rtl) "/" (denr rtl))))
```

Représentation de rationnels abstraits I

— Fractions non réduites
— Réalisation en listes

`(12 18)` et `(2 3)` sont deux représentations correctes du rationnel 2/3.

```
(define numr (lambda (rtl) (car rtl)))

(define denr (lambda (rtl) (cadr rtl)))

(define make-ratl
  (lambda (int1 int2)
    (if (zero? int2)
        (error "make-ratl: The denominator cannot be zero.")
        (list int1 int2))))
```

Avantage : procédures efficaces.
Inconvénient : pas de forme normale unique.

Représentation de rationnels abstraits II

— Fractions non réduites
— Réalisation en paires pointées

`(12 . 18)` et `(2 . 3)` sont deux représentations correctes du rationnel 2/3.

```
(define numr (lambda (rtl) (car rtl)))

(define denr (lambda (rtl) (cdr rtl)))

(define make-ratl
  (lambda (int1 int2)
    (if (zero? int2)
        (error "make-ratl: The denominator cannot be zero.")
        (cons int1 int2))))
```

Remarque. La représentation en paires pointées est plus économique, et donc préférable à la représentation en listes.

Représentation de rationnels abstraits III

— Fractions réduites
— Réalisation en paires pointées

Première technique : changer le *constructeur*

```
(define make-ratl
  (lambda (int1 int2)
    (if (zero? int2)
        (error "make-ratl: The denominator cannot be zero.")
        (let ((g (gcd int1 int2))) (cons (/ int1 g) (/ int2 g))))))
```

Deuxième technique : changer les *accesseurs*

```
(define numr (lambda (rtl) (/ (car rtl) (gcd (car rtl) (cdr rtl)))))
(define denr (lambda (rtl) (/ (cdr rtl) (gcd (car rtl) (cdr rtl)))))
```

L'intervention de `gcd` (réduction) introduit une certaine perte d'efficacité, le plus souvent acceptable.

La première technique (construction lente, accès rapide) est préférable si on accède souvent aux mêmes nombres.

Le type abstrait “polynôme”

Type récursif ; un polynôme comporte : un *degré*, un *coefficient du terme de plus haut degré*, un *reste* (qui est un polynôme).

Le type abstrait “polynôme” comportera donc une constante de base (polynôme nul), un constructeur à trois arguments et trois accesseurs à un argument.

Constante de base : `the-zero-poly`
Un constructeur (trois arguments) : `poly-cons`
Trois accesseurs : `degree`, `lead-coeff`, `rest-poly`

On distinguera les problèmes d’*utilisation* du type “polynôme” et le problème de *réalisation* de ce type. Le second problème revient à programmer la constante de base, le constructeur et les accesseurs.

Polynôme nul et monômes

Reconnaisseur pour la constante de base :

```
(define zero-poly?
  (lambda (poly)
    (and (zero? (degree poly)) (zero? (lead-coef poly)))))
```

Un *monôme* est un polynôme de reste nul :

```
(define make-mono
  (lambda (deg coef) (poly-cons deg coef the-zero-poly)))
```

Le monôme principal d'un polynôme est son monôme de degré le plus élevé (égal au degré du polynôme) :

```
(define lead-mono
  (lambda (poly) (make-mono (degree poly) (lead-coef poly))))
```

Addition de polynômes

```
(define p+
  (lambda (poly1 poly2)
    (cond ((zero-poly? poly1) poly2)
          ((zero-poly? poly2) poly1)
          (else
           (let ((n1 (degree poly1)) (n2 (degree poly2))
                 (a1 (lead-coef poly1)) (a2 (lead-coef poly2))
                 (r1 (rest-poly poly1)) (r2 (rest-poly poly2)))
             (cond ((> n1 n2) (poly-cons n1 a1 (p+ r1 poly2)))
                   ((< n1 n2) (poly-cons n2 a2 (p+ poly1 r2)))
                   (else (poly-cons n1 (+ a1 a2) (p+ r1 r2)))))))))
```

Multiplication de polynômes

```
(define p*
  (letrec
    ((t* (lambda (mono poly)
           (if (zero-poly? poly)
               the-zero-poly
               (poly-cons
                 (+ (degree mono) (degree poly))
                 (* (lead-coef mono) (lead-coef poly))
                 (t* mono (rest-poly poly)))))))
    (lambda (poly1 poly2)
      (if (zero-poly? poly1)
          the-zero-poly
          (p+ (t* (lead-mono poly1) poly2)
              (p* (rest-poly poly1) poly2))))))

(define negative-poly
  (lambda (poly) (p* (make-mono 0 -1) poly)))

(define p-
  (lambda (poly1 poly2) (p+ poly1 (negative-poly poly2))))
```

Evaluation de polynômes

```
(define poly-value
  (lambda (poly num)
    (let ((n (degree poly)))
      (if (zero? n)
          (lead-coef poly)
          (let ((rest (rest-poly poly)))
            (if (< (degree rest) (sub1 n))
                (poly-value
                  (poly-cons (sub1 n) (* num (lead-coef poly)) rest)
                  num)
                (poly-value
                  (poly-cons
                    (sub1 n)
                    (+ (* num (lead-coef poly)) (lead-coef rest))
                    (rest-poly rest))
                  num)))))))
```

Première réalisation (constante, accesseurs)

$2x^3 + 3x - 1$ devient `(2 0 3 -1)`

$2x^{1000} + 3x - 1$ devient `(2` $\underbrace{\texttt{0 .. 0}}_{\text{998 termes !}}$ `3 -1)`

```
(define the-zero-poly '(0))

(define degree (lambda (poly) (sub1 (length poly))))

(define lead-coef (lambda (poly) (car poly)))

(define rest-poly
  (lambda (poly)
    (cond ((zero? (degree poly)) the-zero-poly)
          ((zero? (lead-coef (cdr poly))) (rest-poly (cdr poly)))
          (else (cdr poly)))))
```

Première réalisation (constructeur)

```
(define poly-cons
  (lambda (deg coef poly)
    (let ((dp (degree poly)))
      (cond ((and (zero? deg) (equal? poly the-zero-poly))
             (list coef))
            ((< dp deg)
             (if (zero? coef)
                 poly
                 (cons
                   coef
                   (append (lz (sub1 (- deg dp))) poly))))
            (else
             (error "poly-cons: Degree too high in" poly))))))

(define lz (lambda (n) (if (zero? n) '() (cons 0 (lz (sub1 n))))))
```

Seconde réalisation (constante, accesseurs)

$2x^3 + 3x - 1$ devient `((3 2) (1 3) (0 -1))`

$2x^{1000} + 3x - 1$ devient `((1000 2) (1 3) (0 -1))`

```
(define the-zero-poly '((0 0)))

(define degree (lambda (poly) (caar poly)))

(define lead-coef (lambda (poly) (cadar poly)))

(define rest-poly
  (lambda (poly)
    (if (null? (cdr poly)) the-zero-poly (cdr poly))))
```

Seconde réalisation (constructeur)

```
(define poly-cons
  (lambda (deg coef poly)
    (let ((dp (degree poly)))
      (cond
        ((and (zero? deg) (equal? poly the-zero-poly))
         (list (list deg coef)))
        ((< dp deg)
         (if (zero? coef) poly (cons (list deg coef) poly)))
        (else
         (error "poly-cons: degree too high in" poly))))))
```

Entrée / sortie, conversion I

```
(define digits->poly   ;; liste de coefficients vers polynôme
  (lambda (digit-list)
    (if (null? digit-list)
        (error "digits->poly: Not defined for" digit-list)
        (letrec
          ((make-poly
            (lambda (deg ls)
              (if (null? ls)
                  the-zero-poly
                  (poly-cons deg
                             (car ls)
                             (make-poly (sub1 deg) (cdr ls)))))))
          (make-poly (sub1 (length digit-list)) digit-list)))))
```

Le *résultat affiché* dépend du mode de représentation mais le *texte du programme* n'en dépend pas.

```
(digits->poly '(1 2 3 4)) =1=>  (1 2 3 4)                   ;; mode I
                          =2=>  ((3 1) (2 2) (1 3) (0 4)) ;; mode II
```

Entrée / sortie, conversion II

```
(define poly->digits  ;; polynôme vers liste de coefficients
  (lambda (poly)
    (letrec
      ((convert
         (lambda (p d)
           (cond
             ((zero? d)
              (list (lead-coef p)))
             ((= (degree p) d)
              (cons (lead-coef p) (convert (rest-poly p) (sub1 d))))
             (else
              (cons 0 (convert p (sub1 d))))))))
      (convert poly (degree poly)))))
```

La *donnée entrée* dépend du mode de représentation
mais le *texte du programme* n'en dépend pas.

```
(poly->digits '(1 2 3 4))                    =1=>  (1 2 3 4)
(poly->digits '((3 1) (2 2) (1 3) (0 4))) =2=>  (1 2 3 4)
```

Changement de base numérique I

Conversion entre décimal et “n-aire codé décimal”.

```
(define n-ary->dec
  (lambda (n digits) (poly-value (digits->poly digits) n)))

(define dec->n-ary
  (lambda (n num)
    (letrec
      ((dec->bin
         (lambda (m d)
           (if (zero? m)
               the-zero-poly
               (p+ (make-mono d (remainder m n))
                   (dec->bin (quotient m n) (add1 d)))))))
      (poly->digits (dec->bin num 0)))))

;; 256 = 1*2^8 = 9*27 + 13
(dec->n-ary 2 256)                    (1 0 0 0 0 0 0 0 0)
(n-ary->dec 2 '(1 0 0 0 0 0 0 0 0))   256
(dec->n-ary 27 256)                   (9 13)
(n-ary->dec 27 '(9 13))               256
```

Changement de base numérique II

Conversion de p-aire en q-aire (codés décimal)

```
(define p-ary->q-ary
  (lambda (p q digits)
    (dec->n-ary q (n-ary->dec p digits))))

(p-ary->q-ary 27 2 '(9 13))                    (1 0 0 0 0 0 0 0 0)

(p-ary->q-ary 2 27 '(1 0 0 0 0 0 0 0 0))       (9 13)
```

Autre exemple :

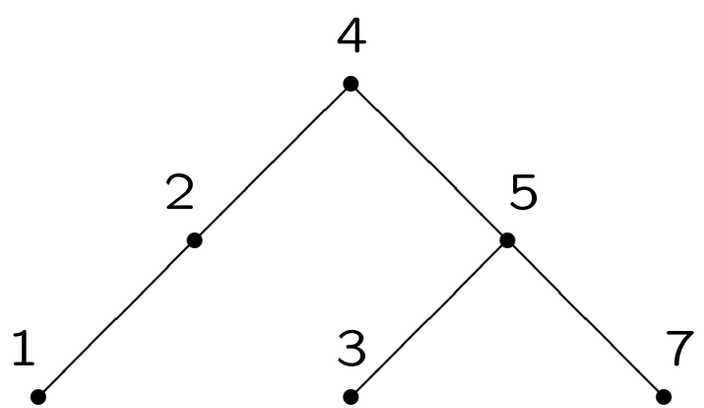
$$5 * 13^3 + 3 * 13^2 + 1 = 2 * 17^3 + 5 * 17^2 + 13 * 17 + 1 = 11493\,.$$

```
(p-ary->q-ary 13 17 '(5 3 0 1))                (2 5 13 1)
```

Arbres binaires complètement étiquetés I

Un K-*arbre binaire complètement étiqueté* est soit l'arbre vide, soit un triplet comportant une *clef* ("key") élément de K, un sous-arbre de gauche et un sous-arbre de droite. On aura donc un constructeur sans argument pour l'arbre vide et un constructeur à trois arguments pour les arbres non vides ; on aura également trois accesseurs.

Exemple, $K = \mathbb{N}$



Représentation concrète simple : liste de trois éléments.

```
(define conc-tree '(4 (2 (1 () ()) ()) (5 (3 () ()) (7 () ()))))
```

Arbres binaires complètement étiquetés II

Constructeurs :

```
(define mk-e-tree (lambda () '()))
(define mk-k-tree (lambda (k l r) (list k l r)))
```

Reconnaisseurs :

```
(define e-tree? null?)
(define k-tree?
  (lambda (x)  ;; x est un objet quelconque
    (and (pair? x) (key? (car x))
         (pair? (cdr x)) (ek-tree? (cadr x))
         (pair? (cddr x)) (ek-tree? (caddr x))
         (null? (cdddr x)))))
(define ek-tree? (lambda (x) (or (e-tree? x) (k-tree? x))))
(define key? (lambda (x) (and (integer? x) (>= x 0))))
```

Accesseurs :

```
(define key car)   (define left cadr)   (define right caddr)
```

Arbres binaires complètement étiquetés III

Représentations concrète et abstraite :

```
(define conc-tree
  '(4 (2 (1 () ()) ()) (5 (3 () ()) (7 () ()))))

(define abst-tree  ;; corriger p. 189
  (mk-k-tree 4
             (mk-k-tree 2
                        (mk-k-tree 1
                                   (mk-e-tree)
                                   (mk-e-tree))
                        (mk-e-tree))
             (mk-k-tree 5
                        (mk-k-tree 3
                                   (mk-e-tree)
                                   (mk-e-tree))
                        (mk-k-tree 7
                                   (mk-e-tree)
                                   (mk-e-tree)))))
```

Arbres binaires complètement étiquetés IV

Présence d'un nombre entier donné dans un $\mathbb{N}$-arbre donné

```
(define in-tree?
  (lambda (i tr)
    (if (e-tree? tr)
        #f
        (or (= i (key tr))
            (in-tree? i (left tr))
            (in-tree? i (right tr))))))
```

Ceci peut se récrire en :

```
(define in-tree?
  (lambda (i tr)
    (and (not (e-tree? tr))
         (or (= i (key tr))
             (in-tree? i (left tr))
             (in-tree? i (right tr))))))
```

Arbres binaires conditionnés I

Un arbre non vide est dit *conditionné* ou *ordonné* si la clef de tout nœud interne est plus grande ou égale aux clefs de tous ses descendants de gauche, et plus petite ou égale aux clefs de tous ses descendants de droite.

Deux écueils à éviter.

L'approche "naïve" : Un arbre non vide serait conditionné si la clef de la racine est comprise entre les clefs des deux fils (s'il existent) et si les deux sous-arbres fils sont eux-mêmes conditionnés. La condition est nécessaire mais pas suffisante (voir exemple) ! ! !

L'approche "prudente" : un arbre non vide serait conditionné si la clef de la racine est supérieure à tous ses descendants de gauche et inférieure à tous ses descendants de droite et si les deux sous-arbres fils sont eux-mêmes conditionnés. La méthode est inefficace, parce que les mêmes comparaisons sont répétées plusieurs fois.

En fait, un arbre est conditionné si ses deux fils sont conditionnés et si sa racine est supérieure à tous les éléments de la branche la plus à droite du fils gauche, et inférieure à tous les éléments de la branche la plus à gauche du fils droit.

Arbres binaires conditionnés II

Les prédicats auxiliaires `greq?` et `leeq?` testent les deux dernières conditions.

```
(define greq?
  (lambda (n tr)
    (or (e-tree? tr) (and (>= n (key tr)) (greq? n (right tr))))))

(define leeq?
  (lambda (n tr)
    (or (e-tree? tr) (and (<= n (key tr)) (leeq? n (left tr))))))

(define condit-1?
  (lambda (tr)
    (or (e-tree? tr)
        (and (condit-1? (left tr)) (greq? (key tr) (left tr))
             (condit-1? (right tr)) (leeq? (key tr) (right tr))))))
```

Ce programme n'est pas optimal ; une version plus efficace est possible si on dispose d'une borne supérieure `*max*` absolue pour les étiquettes des arbres.

Arbres binaires conditionnés III

Version efficace

```
(define condit-2?
  (lambda (tr) (or (e-tree? tr) (tree-ok? 0 tr *max*))))

(define tree-ok?
  (lambda (min tr max)
    (and (<= min (key tr))
         (>= max (key tr))
         (or (e-tree? (left tr)) (tree-ok? min (left tr) (key tr)))
         (or (e-tree? (right tr)) (tree-ok? (key tr) (right tr) max)))))
```

`tr` est un arbre conditionné dont toutes les clefs sont comprises entre les naturels `min` et `max` ssi `(tree-ok? min tr max)` est vrai.

Arbres binaires conditionnés IV

Si un arbre est conditionné, la liste de ses étiquettes est triée, à condition que dans cette liste toute étiquette se trouve entre les étiquettes de ses descendants de gauche et celles de ses descendants de droite.

La fonction `traversal` calcule cette liste :

```
(define traversal
  (lambda (tr)
    (if (e-tree? tr)
        '()
        (append (traversal (left tr))
                (cons (key tr)
                      (traversal (right tr)))))))
```

Arbres binaires conditionnés V

On peut utiliser la technique des accumulateurs pour éviter l'usage de `append`, en écrivant une fonction auxiliaire `trav-a` telle que

`[[(trav-a tr acc)]]`

soit égal à

`[[(append (traversal tr) acc)]]`

```
(define trav-a
  (lambda (tr acc)
    (if (e-tree? tr)
        acc
        (trav-a (left tr)
                (cons (key tr)
                      (trav-a (right tr) acc))))))

(define traversal
  (lambda (tr) (trav-a tr '())))
```

Arbres, tas et tri I

Une arbre est un *tas* si l'étiquette d'un nœud est supérieure aux étiquettes de ses descendants. La notion de tas est utile dans diverses applications. Le programme `heap?` teste si un arbre est un tas (“heap” en anglais) ; il est analogue au programme `condit-2`.

Les règles de portée empêchent toute confusion entre les liaisons locales et globales de `key`, `left` et `right` ; les liaisons locales sont des arbres, les liaisons globales sont des accesseurs.

```
(define heap?
  (lambda (tr)
    (or (e-tree? tr)
        (let ((key (key tr)) (left (left tr)) (right (right tr)))
          (and (greq? key left) (greq? key right)
               (heap? left) (heap? right))))))

(greq? n tr) :
(or (e-tree? tr) (>= n (key tr))
```

Arbres, tas et tri II

Transformation d'un arbre en un tas

```
(define heapify
  (lambda (tr)
    (if (e-tree? tr)
        tr
        (let ((key (key tr)) (left (left tr)) (right (right tr)))
          (let ((lh (heapify left)) (rh (heapify right)))
    (adjust key lh rh))))))

(adjust 9 '(3 () (4 () ())) '(6 () ()))
       (9  (3 () (4 () ()))  (6 () ()))

(adjust 5 '(3 () (4 () ())) '(6 () ()))
       (6  (3 () (4 () ()))  (5 () ()))

(adjust 2 '(3 () (4 () ())) '(6 () ()))
       (6  (3 () (4 () ()))  (2 () ()))
```

Arbres, tas et tri III

```
(define adjust
  (lambda (ky lh rh)
    (cond
      ((and (greq? ky lh) (greq? ky rh)) (mk-k-tree ky lh rh))
      ((greq? ky lh)
       (let ((krh (key rh)) (lrh (left rh)) (rrh (right rh)))
         (mk-k-tree krh lh (adjust ky lrh rrh))))
      ((greq? ky rh)
       (let ((klh (key lh)) (llh (left lh)) (rlh (right lh)))
         (mk-k-tree klh (adjust ky llh rlh) rh)))
      (else
       (let ((klh (key lh)) (krh (key rh)))
         (let ((llh (left lh)) (rlh (right lh))
               (lrh (left rh)) (rrh (right rh)))
           (if (> klh krh)
               (mk-k-tree klh (adjust ky llh rlh) rh)
               (mk-k-tree krh lh (adjust ky lrh rrh)))))))))
```

Arbres, tas et tri IV

Si dans la liste des étiquettes d'un tas, l'étiquette d'un nœud vient toujours avant l'étiquette des descendants de ce nœud, alors la liste est "presque" triée par ordre décroissant. Une variante du prédicat `traversal` permet de le vérifier. Nous écrivons cette variante en utilisant un `letrec` et un accumulateur :

```
(define pre-trav
  (lambda (tr)
    (letrec
      ((pre-trav-a
        (lambda (tr acc)
          (if (e-tree? tr)
              acc
              (cons (key tr)
                    (pre-trav-a (left tr) (pre-trav-a (right tr) acc)))))))
      (pre-trav-a tr '()))))
```

Exercice : spécifier la fonction auxiliaire

Arbres, tas et tri V

Considérons l'arbre dont la représentation concrète est

```
(5 (3 (5 (3 () (5 (4 () (7 () ())) (3 () ()))) (6 () ()))
      (4 (5 (3 () (4 () ())) (6 () ())) ()))
   (6 () (5 (4 () (7 () ())) (3 () ()))))
```

`heapify` transforme cet arbre en le tas

```
(7 (7 (6 (5 () (5 (4 () (3 () ())) (3 () ()))) (3 () ()))
      (6 (5 (4 () (3 () ())) (4 () ())) ()))
   (6 () (5 (5 () (4 () ())) (3 () ()))))
```

qui a même structure ; `pre-trav` fournit les listes

```
(3 4 7 5 3 5 6 3 3 4 5 6 4 5 6 4 7 5 3)
(7 7 6 5 5 4 3 3 3 6 5 4 3 4 6 5 5 4 3)
```

alors que la version triée de ces deux listes est

```
(7 7 6 6 6 5 5 5 5 5 4 4 4 4 3 3 3 3 3)
```

Ceci suggère qu'il devrait exister une variante de `pre-trav` qui, appliquée à un tas, fournirait la liste triée des étiquettes de ce tas.

Arbres, tas et tri VI

```
(define hp-sort-trav  ;; liste triée des étiquettes
  (lambda (hp)
    (if (e-tree? hp)
        '()
        (let ((u1 (hp-sort-trav (left hp)))
              (u2 (hp-sort-trav (right hp))))
          (cons (key hp) (merge u1 u2))))))

(define merge  ;; fusion de deux listes triées
  (lambda (u1 u2)
    (cond ((null? u1) u2)
          ((null? u2) u1)
          (else (let ((a1 (car u1)) (a2 (car u2)))
                  (if (> a1 a2)
                      (cons a1 (merge (cdr u1) u2))
                      (cons a2 (merge u1 (cdr u2)))))))))
```

Cette technique de tri est raisonnablement efficace.

Visualisation de la structure des arbres

```
(define space       ;; (space n) ecrit n blancs
  (lambda (n)
    (if (zero? n)
        (display "")
        (begin (display " ") (space (- n 1))))))

(define pr-tree
  (lambda (tr d)
    (if (e-tree? tr)
        (begin (space d) (display " -"))
        (let ((d1 (1+ d)))
          (begin (space d1) (display (key tr))
                 (newline) (pr-tree (left tr) d1)
                 (newline) (pr-tree (right tr) d1))))))
```

Enregistrements, réalisation concrète I

Les K-arbres binaires complètement étiquetés sont des cas particuliers d'enregistrements. Un *enregistrement* est une structure de donnée admettant un nombre fixé de composants, chacun d'eux ayant un type donné.

> *Le reconnaisseur* `record?` *prend comme arguments un objet* [[u]] *et une liste de propriétés* [[lp]] *et renvoie* `#t` *si* [[u]] *et* [[lp]] *sont des listes de même longueur* ℓ *et si pour tout* $i = 1, \ldots, \ell$*, le* i*ème objet de* [[u]] *satisfait la* i*ème propriété de* [[lp]]*.*

Remarque. Une propriété est ici un prédicat à un argument.

Une solution simple et efficace est

```
(define record?
  (lambda (u lp)
    (or (and (null? u) (null? lp))
        (and (pair? u) (pair? lp) ((car lp) (car u))
             (record? (cdr u) (cdr lp))))))
```

Enregistrements, réalisation concrète II

Le reconnaisseur `k-tree?` introduit plus haut, à savoir

```
(define k-tree?
  (lambda (x)  ;; x est un objet quelconque
    (and (pair? x) (key? (car x))
         (pair? (cdr x)) (ek-tree? (cadr x))
         (pair? (cddr x)) (ek-tree? (caddr x))
         (null? (cdddr x)))))
(define e-tree? null?)
(define ek-tree? (lambda (x) (or (e-tree? x) (k-tree? x))))
(define key? (lambda (x) (and (integer? x) (>= x 0))))
```

pourrait aussi se définir en utilisant `record?` :

```
(define k-tree?
  (lambda (u)
    (record?
      u
      (list (lambda (x) (and (integer? x) (>= x 0)))
            (lambda (v) (or (null? v) (k-tree? v)))
            (lambda (v) (or (null? v) (k-tree? v)))))))
```

Les graphes I

```
(define *g0*
 '((a b c d e f) .
   ((b . a) (b . d) (c . b) (d . c) (d . f) (e . b) (e . f) (f . a))))

(define mk-graph (lambda (nodes arcs) (cons nodes arcs)))

(define nodes (lambda (gr) (car gr)))

(define arcs (lambda (gr) (cdr gr)))

(define mk-arc (lambda (org ext) (cons org ext)))

(define org (lambda (arc) (car arc)))

(define ext (lambda (arc) (cdr arc)))
```

Les graphes II

```
(define *altg2*
  '((a . (b d g h k l o p))
    (b . (c e f p))
    (c . (a e i o))
    (d . ())
    (e . (b k m))
    (f . (c d g m n p))
    (g . (a e j p))
    (h . (b d k l n))
    (i . (b c e m o))
    (j . (c))
    (k . (a f g n p))
    (l . (b k))
    (m . (c e o))
    (n . (a i m))
    (o . (b d e))
    (p . (i j k))))
```

Les graphes III

```
(define nodes
  (lambda (altgr) (map car altgr)))

(define arcs
  (lambda (altgr) (union-map gen-arcs altgr)))

(define union-map
  (lambda (f l)
    (if (null? l)
        '()
        (union (f (car l)) (union-map f (cdr l))))))

(define gen-arcs
  (lambda (nsuccs)
    (let ((n (car nsuccs)) (succs (cdr nsuccs)))
      (map (lambda (x) (mk-arc n x)) succs))))
```

Les graphes IV

```
(mk-graph (nodes *altg2*) (arcs *altg2*))
==>
((a b c d e f g h i j k l m n o p)
 (a . b) (a . d) (a . g) (a . h) (a . k) (a . l) (a . o) (a . p)
 (b . c) (b . e) (b . f) (b . p)
 (c . a) (c . e) (c . i) (c . o)
 (e . b) (e . k) (e . m)
 (f . c) (f . d) (f . g) (f . m) (f . n) (f . p)
 (g . a) (g . e) (g . j) (g . p)
 (h . b) (h . d) (h . k) (h . l) (h . n)
 (i . b) (i . c) (i . e) (i . m) (i . o)
 (j . c)
 (k . a) (k . f) (k . g) (k . n) (k . p)
 (l . b) (l . k)
 (m . c) (m . e) (m . o)
 (n . a) (n . i) (n . m)
 (o . b) (o . d) (o . e)
 (p . i) (p . j) (p . k))
```

Les graphes V

Successeurs d'un nœud dans un graphe

```
(define succs
  (lambda (nd gr)
    (let ((nodes (nodes gr)) (arcs (arcs gr)))
      (if (member nd nodes)
          (succs-arcs nd arcs)
          (error "unknown node" nd)))))

(define succs-arcs
  (lambda (nd arcs)
    (cond ((null? arcs) '())
          ((equal? (org (car arcs)) nd)
           (add-elem (ext (car arcs)) (succs-arcs nd (cdr arcs))))
          (else (succs-arcs nd (cdr arcs))))))

(define add-elem
  (lambda (x l) (if (member x l) l (cons x l))))
```

Les graphes VI

```
(define offspring  ;; Descendance d'un noeud dans un graphe, version naive
  (lambda (nd gr) (off nd gr (length (nodes gr)))))

(define off
  (lambda (nd gr k)
    (if (= k 0) '() (add-elem nd (off* (succs nd gr) gr (- k 1))))))

(define off*
  (lambda (nd* gr k)
    (if (null? nd*)
        '()
        (union (off (car nd*) gr k) (off* (cdr nd*) gr k)))))

(define union
  (lambda (u v)
    (if (null? u) v (add-elem (car u) (union (cdr u) v)))))
```

Les graphes VII

Descendance d'un nœud dans un graphe, deuxième solution

```
(define offspring-bis
  (lambda (nd gr)
    (off*-bis (list nd) gr '())))

(define off*-bis
  (lambda (nd* gr acc)
    (cond ((null? nd*) acc)
          ((member (car nd*) acc)
           (off*-bis (cdr nd*) gr acc))
          (else
           (off*-bis (append (succs (car nd*) gr) (cdr nd*))
                     gr
                     (cons (car nd*) acc))))))
```

Les graphes VIII

Descendance d'un nœud dans un graphe, troisième solution

```
(define offspring-ter
  (lambda (nd gr) (off-ter nd gr '())))

(define off-ter
  (lambda (nd gr acc)
    (if (member nd acc) acc (off*-ter (succs nd gr) gr (cons nd acc)))))

(define off*-ter
  (lambda (nd* gr acc)
    (cond ((null? nd*) acc)
          ((member (car nd*) acc)
           (off*-ter (cdr nd*) gr acc))
          (else
           (off-ter (car nd*)
                    gr
                    (off*-ter (cdr nd*) gr acc))))))
```

Les graphes IX

Essais.

Dans [[*g0*]], on a les arcs

```
c->b  b->a
      b->d d->c
           d->f f->a
```

donc la descendance de [[c]] dans [[*g0*]] comporte

[[a]], [[b]], [[c]], [[d]] et [[f]].

On a effectivement

```
(offspring     'c *g0*) (c b d f a)

(offspring-bis 'c *g0*) (f d a b c)

(offspring-ter 'c *g0*) (a f d b c)
```

11. Un exercice de programmation

Buts :

- *Raisonnement récursif*. Le type des arguments suggère souvent un schéma de récursion approprié.
- *Programmation "top-down"* . On définit d'abord la procédure principale, puis les procédures auxiliaires s'il y a lieu.
- *Abstraction sur les données*.

Enoncé. On a une collection d'objets ; chaque objet a un *poids* (naturel non nul) et une *utilité* (réel strictement positif). On se donne aussi un *poids maximal* (nombre naturel). Un *chargement* est une sous-collection d'objets ; le poids d'un chargement est naturellement la somme des poids des objets qu'il contient ; son utilité est la somme des utilités.
Le problème consiste à déterminer le chargement d'utilité maximale, dont le poids n'excède pas le poids maximal.

Remarques. Le problème du "sac à dos" (*knapsack*) est NP-complet.
Il a des applications en cryptographie.

Stratégie : récursivité structurelle (mixte)

L'idée algorithmique utile ici est d'application fréquente dans les problèmes combinatoires. Elle consiste simplement à répartir les entités à considérer ou à dénombrer en deux classes, que l'on traite séparément (appels récursifs), puis à combiner les deux résultats partiels. On rappelle d'abord trois exemples classiques.

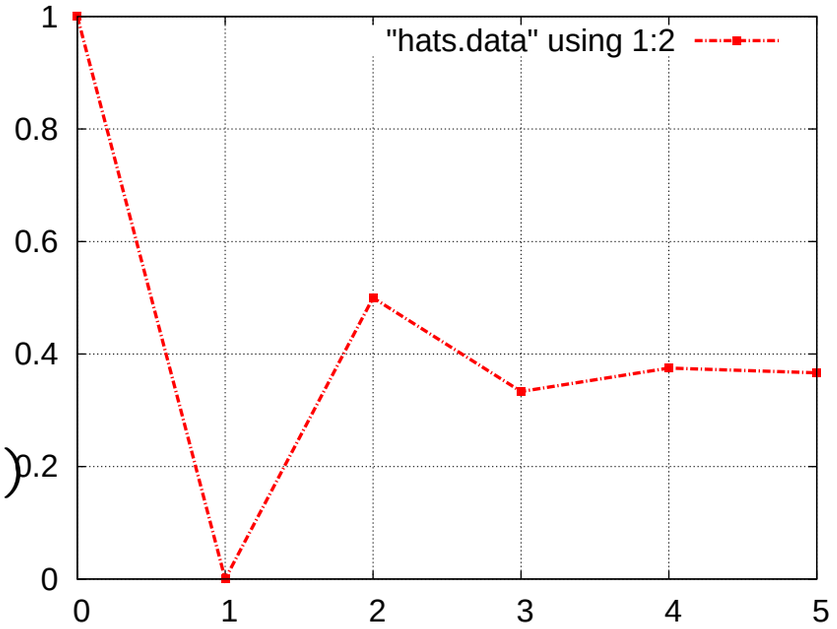
- Nombre $C(n,k)$ de choix de k objets parmi n ($0 \leq k \leq n$) ?

Cas de base, $k = 0$ ou $k = n$, $C(n,k) = 1$
Cas inductif, $0 < k < n$, soit X un objet
X inclus : $C(n-1,k-1)$
X exclu : $C(n-1,k)$
total : $C(n-1,k-1) + C(n-1,k)$.

- Nombre de partages de n objets distincts en k lots non vides ?

Cas de base, $k = n$, $P(n,k) = 1$
$k = 0 < n$, $P(n,k) = 0$
Cas inductif, $0 < k < n$, soit X un objet
X isolé : $P(n-1,k-1)$
X non isolé : $k\,P(n-1,k)$
total : $P(n-1,k-1) + k\,P(n-1,k)$.

Les dérangements



- Combien de "dérangements" de $(1, 2, \ldots, n)$?
  (On doit avoir $p(i) \neq i$, pour tout i.)
  Cas de base, $n = 0$, $D(n) = 1$
  $n = 1$, $D(n) = 0$
  Cas inductif, $n \geq 2$, soit $p(n) = i$ $(i \in \{1, \ldots, n-1\})$.
  $p(i) = n$: $(n-1)\, D(n-2)$ cas
  $p(i) \neq n$: $(n-1)\, D(n-1)$ cas
  total : $(n-1)\,(D(n-2) + D(n-1))$.

Problème des chapeaux. Si n personnes mélangent leurs chapeaux puis se les réattribuent au hasard, la probabilité que personne ne récupère le sien est :

$$P(n) = D(n)/n!$$

On note une convergence rapide vers $e^{-1} = 0.367879...$ (voir graphique).

Stratégie récursive mixte pour le problème “sac à dos”

Cas de base.
La collection est vide ou le poids maximal est nul.
La solution est le chargement vide, d'utilité nulle.

Cas inductif.
La collection C n'est pas vide et le poids maximal L est strictement positif.

Par rapport à un objet arbitraire X de la collection, il y a deux types de chargements : ceux qui négligent X (type I) et ceux qui contiennent X (type II). Un chargement de type I est relatif à la collection $C \setminus \{X\}$ et au poids maximal L. Un chargement de type II comporte X, plus un chargement relatif à la collection $C \setminus \{X\}$ et au poids maximal $L - p(X)$; le type II n'existe pas si $L < p(X)$.

La tactique est donc de calculer, séparément, les deux solutions optimales, relatives à une collection amputée d'un élément (parfois une seule, si $L < p(X)$), puis d'en déduire le chargement optimal pour la collection donnée.

Types abstraits de données

Le type “collection” est récursif. On a

— La constante de base `the-empty-coll` ;

— Le constructeur `add-obj-coll` (deux arguments) ;

— Les reconnaisseurs `coll?` et `empty-coll?` ;

— Les accesseurs `obj-coll` et `rem-coll`.

Pour le type non récursif “objet”, on a

— Le constructeur `mk-obj` (deux arguments) ;

— Le reconnaisseur `obj?` ;

— Les accesseurs `poids` et `utilite`.

On écrit facilement les relations algébriques induites par ces définitions. Par exemple, dans un environnement où `x` et `c` sont liés respectivement à un objet et à une collection, si `(empty-coll? c)` a la valeur `#f`, les expressions `c` et `(add-obj-coll (obj-coll c) (rem-coll c))` ont même valeur.

On utilise aussi un type `solution` ; un objet de ce type comporte une collection avec son poids total et son utilité totale ; on aura notamment le constructeur `mk-sol` et les trois accesseurs `char`, `ptot` et `utot`.

Développement du programme I

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (empty-coll? c)) ;; double cas de base
        (mk-sol the-empty-coll 0 0)
        (...))))
```

La partie à préciser concerne le cas inductif. Son traitement requiert la distinction d'un objet `x` de `c` et, au moins, le calcul récursif d'une solution de type I. On obtient

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c))
               (rc (rem-coll c))
               (s1 (knap pm rc)))
          (...)))))
```

Développement du programme II

Pour savoir si on devra aussi considérer une solution de type II, il faut comparer le poids de `x` au poids maximal ; on a

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c)) (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x)) (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (...)
              s1)))))
```

Développement du programme III

Il reste à déterminer si la solution cherchée est `s1`, ou la solution obtenue en ajoutant `x` à `s2`.

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c))
               (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x))
               (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (if (> (+ (utot s2) (utilite x)) (utot s1))
                    (mk-sol (add-obj-coll x (char s2))
                            (+ px (ptot s2))
                            (+ ux (utot s2)))
                    s1))
              s1)))))
```

Version finale

Il reste à déterminer si la solution cherchée est `s1`, ou la solution obtenue en ajoutant `x` à `s2`.

```
(define knap
  (lambda (pm c)
    (if (or (= 0 pm) (null? c))
        (mk-sol the-empty-coll 0 0)
        (let* ((x (obj-coll c)) (rc (rem-coll c))
               (s1 (knap pm rc))
               (px (poids x)) (ux (utilite x)))
          (if (>= pm px)
              (let ((s2 (knap (- pm px) rc)))
                (if (> (+ (utot s2) (utilite x)) (utot s1))
                    (mk-sol (add-obj-coll x (char s2))
                            (+ px (ptot s2))
                            (+ ux (utot s2)))
                    s1))
              s1)))))
```

Structures de donnée

On peut réaliser le type abstrait `collection` par le type `list`, avec les correspondances suivantes :

`the-empty-coll`	`'()`
`add-obj-coll`	`cons`
`coll?  empty-coll?`	`list?  null?`
`obj-coll  rem-coll`	`car  cdr`

Le type `objet` est concrétisé par le type `pair` :

`mk-obj`	`cons`
`obj?`	`pair?`
`poids  utilite`	`car  cdr`

Pour le type `solution`, on utilise aussi le type `pair`, restreint au cas où la deuxième composante est aussi de type `pair` :

`(mk-sol c p u)`	`(cons c (cons p u))`
`char  ptot  utot`	`car  cadr  cddr`

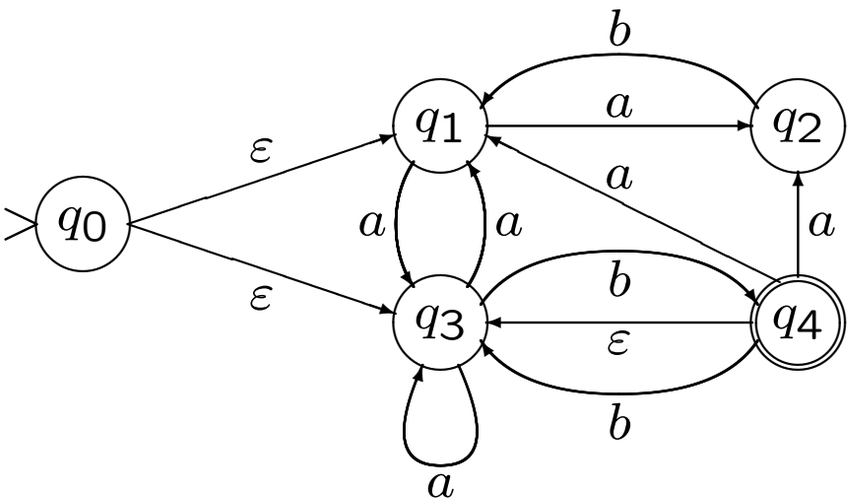
Essais

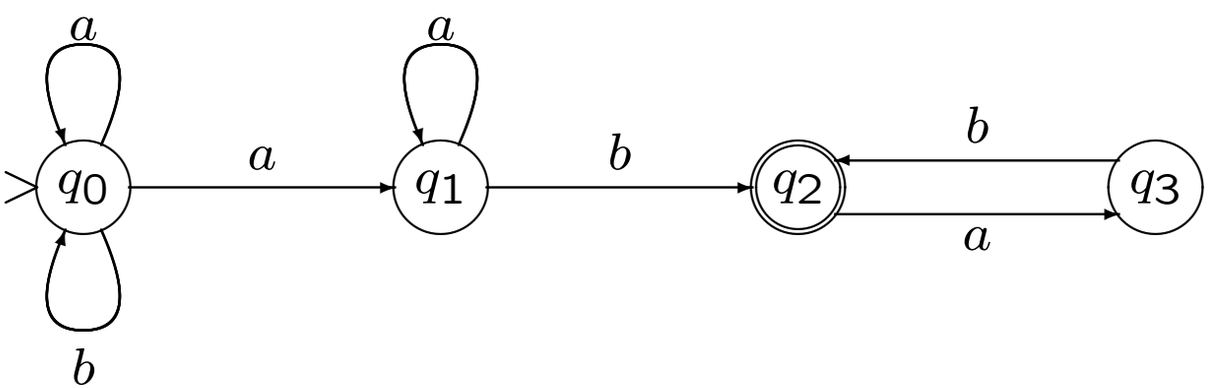
```
c       '((4 . 9) (3 . 8) (2 . 4) (1 . 1) (2 . 3)
          (6 . 9) (5 . 5) (4 . 8) (1 . 2) (2 . 1)
          (1 . 2) (1 . 1) (7 . 9) (6 . 6) (5 . 4)
          (4 . 5) (3 . 2) (2 . 3) (2 . 2) (3 . 3)))

(knap 0 c)       (()                                          0 . 0)

(knap 5 c)       ((3 . 8) (1 . 2) (1 . 2))                    5 . 12)

(knap 10 c)      (((4 . 9) (3 . 8) (2 . 4) (1 . 2))           10 . 23)

(knap 15 c)      (((4 . 9) (3 . 8) (2 . 4) (4 . 8)
                   (1 . 2) (1 . 2))                           15 . 33)

(knap 20 c)      (((4 . 9) (3 . 8) (2 . 4) (6 . 9)
                   (4 . 8) (1 . 2))                           20 . 40)

(knap 25 c)      (((4 . 9) (3 . 8) (2 . 4) (2 . 3) (6 . 9) (4 . 8)
                   (1 . 2) (1 . 2) (2 . 3))                   25 . 48)
```

v	5	10	15	20	25
Temps	1	6	32	141	317

Le comportement est typiquement exponentiel.

Déterminisation d'un automate I





Déterminisation d'un automate II

```
(define *autom01*
  (list->automaton
    '((q0 q1 q2 q3 q4)
      (a b)
      ((q0 () q1)  (q0 () q3)  (q1 (a) q2) (q1 (a) q3)
       (q2 (b) q1) (q3 (a) q1) (q3 (a) q3) (q3 (b) q4)
       (q4 (a) q1) (q4 (a) q2) (q4 () q3)  (q4 (b) q3))
      q0
      (q4))))

(define *autom02*
  (list->automaton
    '((q0 q1 q2 q3)
      (a b)
      ((q0 (a) q0) (q0 (b) q0) (q0 (a) q1) (q1 (a) q1)
       (q1 (b) q2) (q2 (a) q3) (q3 (b) q2))
      q0
      (q2))))
```

Déterminisation d'un automate III

Réalisation du type automate

```
(define list->automaton (lambda (x) x))

(define mk-aut list)          (define mk-trans list)

(define states car)           (define orig car)
(define alph cadr)            (define word cadr)
(define trans caddr)          (define extr caddr)
(define init cadddr)
(define finals caddddr)

(define caddddr (lambda (x) (car (cddddr x))))
```

Déterminisation d'un automate IV

```
(define filter
  (lambda (p? l)
    (cond ((null? l) '())
          ((p? (car l))
           (cons (car l) (filter p? (cdr l))))
          (else (filter p? (cdr l))))))

(define map-filter
  (lambda (f p? l)
    (if (null? l)
        '()
        (let ((rec (map-filter f p? (cdr l))))
          (if (p? (car l))
              (cons (f (car l)) rec)
              rec)))))
```

Déterminisation d'un automate V

```
(define extend
  (lambda (aut q)
    (let ((states (states aut))
          (trans (trans aut)))
      (let ((arcs
             (map-filter
               (lambda (tr)
                 (mk-arc (orig tr) (extr tr)))
               (lambda (tr)
                 (null? (word tr)))
               trans)))
        (offspring-ter q
                       (mk-graph states arcs))))))

(define s-extend
  (lambda (aut q)
    (sort (extend aut q))))
```

Déterminisation d'un automate VI

```
(define union*
  (lambda (l)
    (if (null? l) '() (union (car l) (union* (cdr l))))))

(define extend*
  (lambda (aut q*) (union* (map (lambda (q) (extend aut q)) q*))))

(define next
  (lambda (aut q x)
    (let ((trans (trans aut)))
      (map-filter
        extr
        (lambda (tr)
          (and (equal? (orig tr) q) (equal? (word tr) (list x))))
        trans))))

(define next*
  (lambda (aut q* x) (union* (map (lambda (q) (next aut q x)) q*))))
```

Déterminisation d'un automate VII

```
(define del (lambda (aut q* x) (extend* aut (next* aut q* x))))

(define s-del (lambda (aut q* x) (sort (del aut q* x))))

(define determinize
  (lambda (aut)
    (let ((states (states aut)) (alph (alph aut)) (trans (trans aut))
          (init (init aut)) (finals (finals aut)))
      (let ((new-states (subsets states)))
        (mk-aut new-states
                alph
                (union-map
                  (lambda (x)
                    (s-map (lambda (ns) (mk-trans ns x (s-del aut ns x)))
                           new-states))
                  alph)
                (s-extend aut init)
                (filter (lambda (ns) (inter? finals ns)) new-states))))))
```

Déterminisation d'un automate VIII

```
(define minimize
  (lambda (aut)
    (let ((states (states aut)) (alph (alph aut)) (trans (trans aut))
          (init (init aut)) (finals (finals aut)))
      (let ((graph
             (mk-graph states
                       (s-map (lambda (tr) (mk-arc (orig tr) (extr tr)))
                              trans))))
        (let ((m-states (offspring-ter init graph)))
          (let ((m-trans
                 (filter (lambda (tr) (in? (orig tr) m-states)) trans))
                (m-finals
                 (filter (lambda (ns) (in? ns m-states)) finals)))
            (mk-aut m-states alph m-trans init m-finals)))))))
```

Déterminisation d'un automate IX

```
(define *autom01*     ;; fig. 10.3, p. 218 (El. Prog)
  '((q0 q1 q2 q3 q4)
    (a b)
    ((q0 () q1) (q0 () q3) (q1 (a) q2) (q1 (a) q3)
     (q2 (b) q1) (q3 (a) q1) (q3 (a) q3) (q3 (b) q4)
     (q4 (a) q1) (q4 (a) q2) (q4 () q3) (q4 (b) q3))
    q0
    (q4)))

(define *det01* (determinize *autom01*))

(define *small01* (minimize *det01*))

(define *autom02*     ;; ex. 2.6, p. 29 (Calc)
  '((q0 q1 q2 q3)
    (a b)
    ((q0 (a) q0) (q0 (b) q0) (q0 (a) q1)
     (q1 (a) q1) (q1 (b) q2) (q2 (a) q3) (q3 (b) q2))
    q0
    (q2)))

(define *det02* (determinize *autom02*))

(define *small02* (minimize *det02*))
```

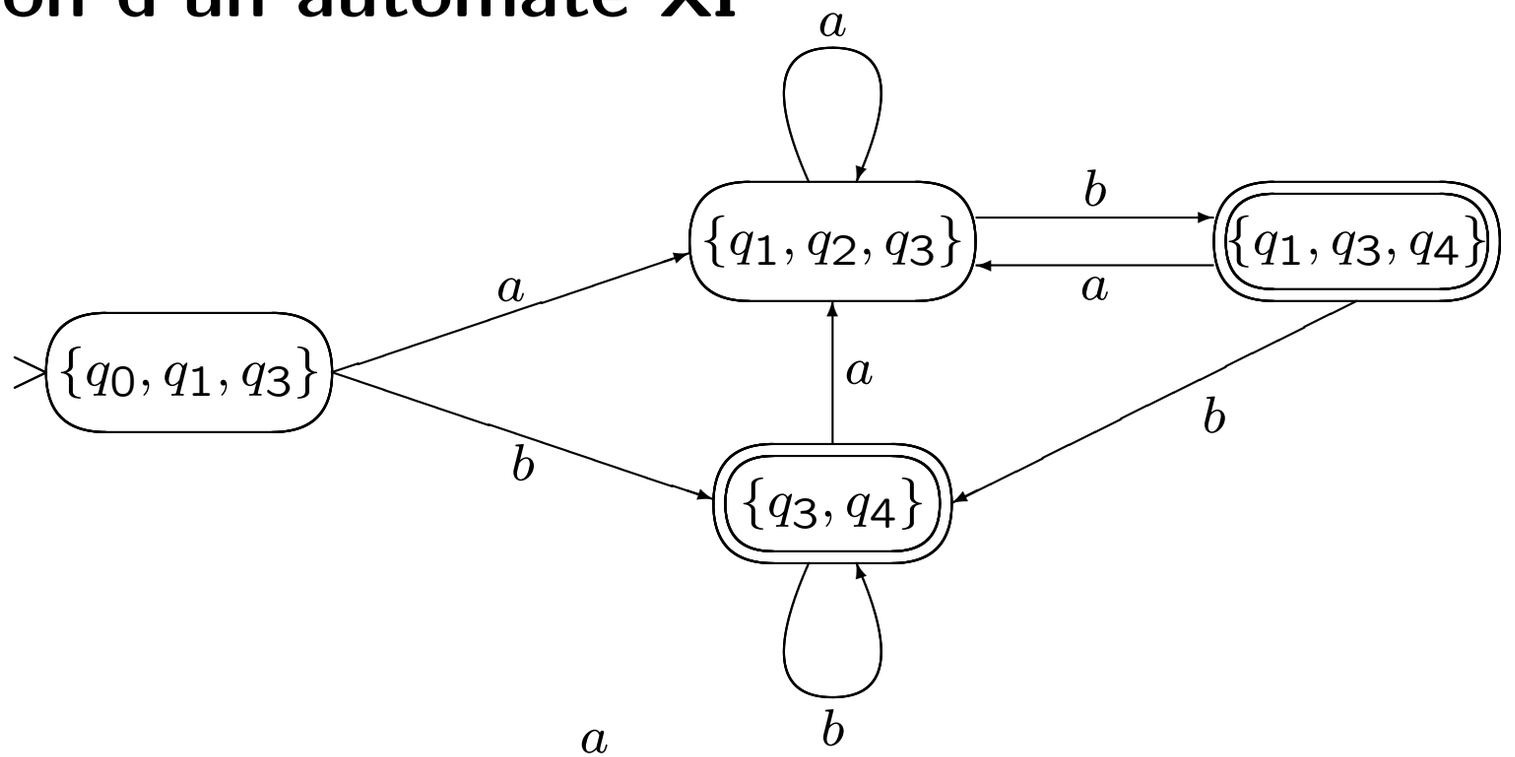
Déterminisation d'un automate X

Si un automate non déterministe comporte n états, et si l'alphabet contient p symboles, l'automate déterministe correspondant comportera 2^n états et $p\,2^n$ transitions. Pour les deux exemples, on a respectivement 32 et 16 états, et 64 et 32 transitions. Les versions minimisées sont nettement plus petites, avec seulement 4 états et donc 8 transitions :

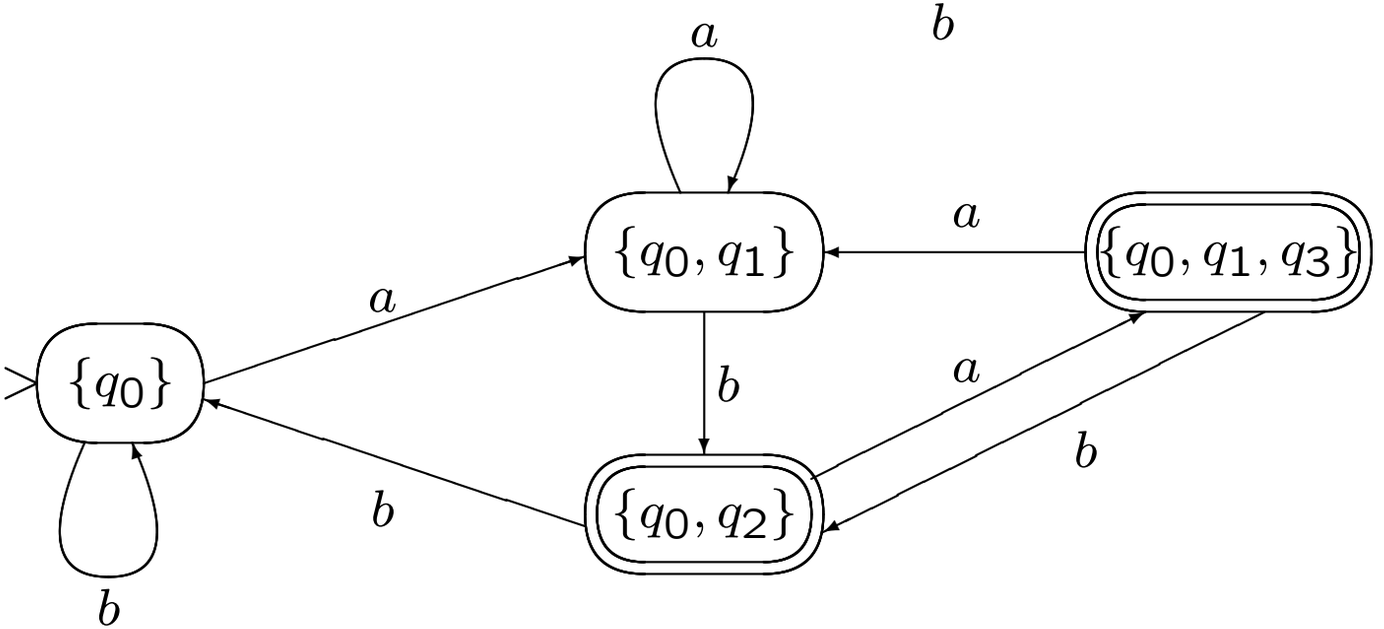
```
(automaton->list *small01*) ==>
(((q1 q3 q4) (q1 q2 q3) (q3 q4) (q0 q1 q3))
 (a b)
 (((q3 q4) a (q1 q2 q3)) ((q1 q3 q4) a (q1 q2 q3))
  ((q1 q2 q3) a (q1 q2 q3)) ((q0 q1 q3) a (q1 q2 q3))
  ((q3 q4) b (q3 q4)) ((q1 q3 q4) b (q3 q4))
  ((q1 q2 q3) b (q1 q3 q4)) ((q0 q1 q3) b (q3 q4)))
 (q0 q1 q3)
 ((q3 q4) (q1 q3 q4)))

(automaton->list *small02*) ==>
(((q0 q1 q3) (q0 q2) (q0 q1) (q0))
 (a b)
 (((q0) a (q0 q1)) ((q0 q2) a (q0 q1 q3)) ((q0 q1) a (q0 q1))
  ((q0 q1 q3) a (q0 q1)) ((q0) b (q0)) ((q0 q2) b (q0))
  ((q0 q1) b (q0 q2)) ((q0 q1 q3) b (q0 q2)))
 (q0)
 ((q0 q2)))
```

Déterminisation d'un automate XI

small01



small02



12. Abstraction procédurale

Principe. La notion de procédure est la clef de la décomposition d'un problème en sous-problèmes. Le fait qu'en Scheme une procédure puisse accepter des procédures comme données et produire des procédures comme résultats rend le langage spécialement adapté à *l'abstraction procédurale*.

Conséquence. En programmation comme en mathématique, il est souvent opportun de reconnaître en un problème donné un cas particulier d'un problème plus général, et même de chercher d'emblée à résoudre le problème général, ce qui produira une procédure largement réutilisable dans des contextes variés.

Application. On va voir comment une procédure itérative de calcul de la racine carrée peut se généraliser en une procédure très générale de mise en œuvre d'un processus d'approximation.

Méthode itérative de calcul de $\sqrt{x}$

$y_0 = 1 \qquad y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$

Si $x = 2$: $1\,,\ \frac{1}{2}\left(1 + \frac{2}{1}\right) = 1.5\,,\ \frac{1}{2}\left(1.5 + \frac{2}{1.5}\right) = 1.4167\,,\ \ldots$

```
(define sqrt-iter
  (lambda (guess x)
    (display "  ") (write guess)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x))))

(define improve
  (lambda (guess x) (/ (+ guess (/ x guess)) 2)))

(define good-enough?
  (lambda (guess x) (< (abs (- (square guess) x)) 0.0000000001)))

(sqrt-iter 1.0 2.0)
1.  1.5  1.4166666666666665  1.4142156862745097  1.4142135623746899
;Value: 1.4142135623746899
```

Abstraction et généralisation I

Généralisation élémentaire : passer de la racine carrée à la racine $p$ième.

Méthode itérative de calcul de $\sqrt[p]{x}$.

$y_0 = 1 \qquad y_{n+1} = \frac{1}{p}\left((p-1)y_n + \frac{x}{y_n^{p-1}}\right)$

```
(define sqrt-p-iter
  (lambda (guess x p) ...

(define improve
  (lambda (guess x p)
    (/ (+ (* (- p 1) guess)
          (/ x (expt guess (- p 1)))) p)))

(define good-enough?
  (lambda (guess x p)
    (< (abs (- (expt guess p) x)) 0.00000001)))

(sqrt-p-iter 1.0 729.0 3)   9.
(sqrt-p-iter 1.0 2.0 2)     1.4142135623746899
```

Abstraction et généralisation II

Généralisation moins élémentaire :
passer de l'équation $y^p - x = 0$ à l'équation $f(y) = 0$.

$y_0 = 1 \qquad y_{n+1} = y_n - \frac{f(y_n)}{Df(y_n)}$

```
(define solve
  (lambda (guess f Df)
    (if (good-enough? guess f)
        guess
        (solve (improve guess f Df) f Df))))

(define improve
  (lambda (guess f Df) (- guess (/ (f guess) (Df guess)))))

(define good-enough?
  (lambda (guess f) (< (abs (f guess)) 0.1)))

(solve 1.0 (lambda (y) (- (expt y 3) 729.0))
           (lambda (y) (* 3 (expt y 2))))       9.0000220253469
```

Abstraction et généralisation III

Résolution itérative de $f(y) = 0$ avec calcul approximatif de la dérivée

```
(define newton
  (lambda (gu f dx)
    (if (good-enough? gu f)
        gu
        (newton (improve gu f dx) f dx))))

(define deriv
  (lambda (f dx) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))

(define improve
  (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))

(define good-enough?
  (lambda (gu f) (< (abs (f gu)) 0.1)))

(newton 1.0 (lambda (y) (- (expt y 3) 729.0)) 0.0001)  9.000022153425999
(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)         0.750367529858334
(cos 0.750367529858334)                                0.731438296864949
```

(Ici, `good-enough?` ... ne mérite pas son nom.)

Abstraction et généralisation IV

```
(define newton
  (lambda (gu f dx)
    (let* ((deriv
            (lambda (f dx)
              (lambda (x) (/ (- (f (+ x dx)) (f x)) dx))))
           (improve
            (lambda (gu f dx) (- gu (/ (f gu) ((deriv f dx) gu)))))
           (good-enough?
            (lambda (gu f) (< (abs (f gu)) 0.001))))
      (if (good-enough? gu f)
          gu
          (newton (improve gu f dx) f dx)))))
```

```
(newton 1.0 (lambda (y) (- y (cos y))) 0.0001)          0.7391131535431725
(cos 0.7391131535431725)                                0.7390662580950105
```

(good-enough? a été modifié.)

Abstraction et généralisation V

Calcul itératif de point fixe : résoudre $x = f(x)$

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
           (lambda (gu f) (< (abs (- gu (f gu))) 0.001))))
      (if (good-enough? gu f) gu (fixpoint (f gu) f)))))

(fixpoint 1.0 cos)                      0.739567202212561
(cos 0.739567202212561)                 0.738760319874213
(fixpoint 1.0 (lambda (x) (/ 2 x)))     ...
```

Amélioration par lissage

```
(define fixpoint
  (lambda (gu f)
    (let ((good-enough?
           (lambda (gu f) (< (abs (- gu (f gu))) 0.001)))
          (improve
           (lambda (gu f) (/ (+ gu (f gu)) 2))))
      (if (good-enough? gu f) gu (fixpoint (improve gu f) f)))))

(fixpoint 1.0 (lambda (x) (/ 2 x)))     1.414...
```

Abstraction et généralisation VI

Idée : `fixpoint` et `newton` sont deux instances de `iterative-improve`. `iterative-improve` prend comme arguments des procédures `good-enough?` et `improve` et renvoie comme valeur une procédure `f` telle que `(f gu)` soit `(if (good-enough? gu) gu (f (improve gu)))`

On doit donc

- définir `iterative-improve`
- écrire `fixpoint` et `newton` comme instances de `iterative-improve`

Ceci permettra des appels tels que

```
((fixpoint cos) 1.0)                                  0.7392146118880453
((newton (lambda (x) (- x (cos x))) 0.001) 1.0)       0.7391155232281558
```

Abstraction et généralisation VII

```
(define iterative-improve
  (lambda (good-enough? improve)
    (lambda (gu)
      (letrec
        ((f (lambda (g) (if (good-enough? g) g (f (improve g))))))
        (f gu)))))

(define fixpoint
  (lambda (f)
    (iterative-improve (lambda (gu) (< (abs (- gu (f gu))) 0.001))
                       (lambda (gu) (/ (+ gu (f gu)) 2)))))

(define newton
  (lambda (f dx)
    (let ((deriv
           (lambda (f) (lambda (x) (/ (- (f (+ x dx)) (f x)) dx)))))
      (iterative-improve
        (lambda (gu) (< (abs (- gu (f gu))) 0.001))
        (lambda (gu) (- gu (/ (f gu) ((deriv f) gu))))))))
```

Itérateur I

On appelle $n$ième itérée de la fonction f de D dans D
la composée de n fonctions égales à f.

> *Ecrire une fonction* `iter`
> *tel que pour tout naturel* `n`,
> `(iter n)` *soit la fonction*
> *qui à toute fonction* `f` *auto-composable*
> *associe la* `n`*ième itérée de* `f`*.*

La solution est immédiate, mais il faut veiller à respecter le type fonctionnel des objets manipulés.

```
(define iter                                    ;; On définit iter
  (lambda (n)                                   ;; une fonction à un argument
    (lambda (f)                                 ;; (iter n) associe à f
      (if (zero? n)                             ;; si n vaut 0
          (lambda (x) x)                        ;; la fonction identité
          (compose f ((iter (- n 1)) f))))))    ;; sinon la composée de ...
```

Itérateur II

```
(define iter      ;; variante
  (lambda (n)
    (lambda (f)
      (lambda (x)
        (if (zero? n) x (f (((iter (- n 1)) f) x)))))))

(define it        ;; scinder la difficulté
  (lambda (n f x)
    (if (= n 0) x (f (it (- n 1) f x)))))

(define iter-bis  ;; autre variante, équivalente
  (lambda (n)
    (lambda (f) (lambda (x) (it n f x)))))

(((iter     3) cos) 1)   .654289790497791
(((iter-bis 3) cos) 1)   .654289790497791
(cos (cos (cos 1)))      .654289790497791
(it 3 cos 1)             .654289790497791
```

13. Abstraction procédurale, exemple

On résout deux problèmes classiques,

— le problème des 8 reines,

— le problème du voyageur de commerce.

On voit qu'en dépit des différences apparentes, ces deux problèmes sont du type "recherche et optimisation", puis on montre que les programmes les résolvant sont des instances d'un schéma très général, ce qui amène à résoudre

— le problème de la recherche abstraite.

On donne en même temps de nouvelles applications de la récursion, de la structuration en procédures et de l'usage de la forme `let` et de ses variantes.

On travaille en données concrètes et on laisse à l'étudiant le problème de la réécriture avec données abstraites.

Introduction au problème des 8 reines I

Un échiquier est un carré de 8 cases de côté.
Deux reines sur un échiquier sont en prise si elles se trouvent sur une même horizontale, verticale ou diagonale.
Si les positions des reines sont (a, b) et $(a + i, b + j)$, les reines sont donc en prise si $i = 0$, $j = 0$ ou $i = \pm j$.
Il est clair que l'on ne peut placer plus de huit reines sur un échiquier sans que deux d'entre elles soient en prise.
Le problème posé consiste à construire toutes les possibilités de placer huit reines sur un échiquier sans que deux d'entre elles soient en prise.
Une solution (s'il en existe) comportera une reine par colonne, chacune se trouvant sur une ligne distincte.
On pourra la représenter par une liste de type $(a_1, \ldots, a_8)$, permutation de la liste $(1, \ldots, 8)$. Les positions correspondantes des reines forment l'ensemble $\{(a_i, i) : i = 1, \ldots, 8\}$.
Toute solution est une permutation ; une permutation vérifie naturellement les conditions relatives aux horizontales et aux verticales, et sera une solution si elle vérifie en outre la condition relative aux diagonales.

Introduction au problème des 8 reines II

On pourrait penser à l'approche récursive habituelle, et exprimer les solutions du problème des n reines en fonction de celles du problème des $n-1$ reines. Il apparaît cependant que cette approche n'est pas facile. On observe d'ailleurs que le problème admet une solution pour $n = 1$, aucune solution pour $n = 2$ et $n = 3$, et deux solutions pour $n = 4$.

Une approche alternative naturelle est de construire les $n!$ permutations représentant tous les "candidats-solutions" (ceux-ci respectent les conditions horizontales et verticales, mais pas nécessairement les conditions diagonales) et ensuite de "filtrer" ceux-ci pour ne retenir que les (vraies) solutions. Cette approche est facile à mettre en œuvre mais très inefficace. Pour $n = 8$, il y a en effet $8! = 40\,320$ candidats-solutions mais seulement 92 solutions.

Introduction au problème des 8 reines III

Il suffit d'imaginer une mise en œuvre concrète, à la main, pour découvrir une optimisation élémentaire. L'expérimentateur place les reines une à une, dans des colonnes contigues. S'il place, par exemple, une reine en colonne 1, ligne 4, puis une seconde en colonne 2, ligne 5, la condition diagonale est déjà violée, et l'adjonction de nouvelles reines dans les six autres colonnes n'arrangera rien. Les $6! = 720$ candidats-solutions correspondants peuvent être éliminés d'un coup.

Le but de l'exercice est de montrer comment l'usage d'une stratégie clairement procédurale peut s'intégrer méthodiquement dans le paradigme fonctionnel. D'autre part, la stratégie "generate-and-test" et ses diverses variantes sont intéressantes pour elles-mêmes, vu leurs applications nombreuses et importantes.

Le problème des 8 reines I

On représentera naturellement un placement de reines sur l'échiquier par une liste. On considèrera par exemple que, si $n = 8$, la liste `(7 3 1)` identifie un échiquier comportant des reines en 6ième, 7ième et 8ième colonnes, placées respectivement sur les lignes 7, 3 et 1. Comme un élément s'ajoute commodément en tête de liste, on considèrera que l'échiquier est rempli de la droite (col 8) vers la gauche (col 1).

Le placement `(7 3 1)` est *légal* car la condition diagonale est respectée. Un placement légal est une *configuration*. Une première fonction à réaliser est le prédicat `legal?` tel que `(legal? try conf)` soit vrai si l'adjonction, dans la colonne libre la plus à droite de la configuration `conf`, d'une reine en ligne `try`, fournit encore une configuration.

On ne peut pas ramener le cas `(legal? try conf)` au cas `(legal? try (cdr conf))` parce qu'il y a décalage d'une colonne pour le placement de la nouvelle reine. Pour appliquer cette récursion habituelle, il faut tenir compte de ce décalage, soit d. Pour ce faire, on définit un prédicat auxiliaire `good?` ; on pourrait, avec des notations évidentes, ramener le cas `(good? try conf d)` au cas `(good? try (cdr conf) (add1 d))` mais on préfère introduire deux paramètres, `up` et `down`, correspondant respectivement à `(+ try d)` et `(- try d)`.

Le problème des 8 reines II

```
(define legal?
  (lambda (try conf) (good? try conf (add1 try) (sub1 try)))))

(define good?
  (lambda (try conf up down)
    (or (null? conf)
        (let ((next-pos (car conf)))
          (and (not (= next-pos try))
               (not (= next-pos up))
               (not (= next-pos down))
               (good? try (cdr conf) (add1 up) (sub1 down)))))))
```

La fonction auxiliaire `good?` peut être rendue locale à la fonction principale `legal?`. Comme la définition de `good?` est récursive, il conviendra d'utiliser `letrec`. Puisque le premier argument de `good?` ne varie pas lors des appels récursifs, on peut l'omettre.

A titre d'exercice, le lecteur pourra préciser la spécification de `legal?` donnée précédemment, et spécifier la fonction auxiliaire `good?`.

Le code de `legal?` est donné page suivante ; le prédicat `solution?` détermine si une configuration est une solution.

Le problème des 8 reines III

L'adjonction de la reine `try` à la configuration `conf` produit-elle une configuration ?

```
(define legal?
  (lambda (try conf)
    (letrec
      ((good?
         (lambda (new-pl up down)
           (or (null? new-pl)
               (let ((next-pos (car new-pl)))
                 (and (not (= next-pos try))
                      (not (= next-pos up))
                      (not (= next-pos down))
                      (good? (cdr new-pl) (add1 up) (sub1 down))))))))
      (good? conf (add1 try) (sub1 try)))))

(legal? 2 '(4 8))   #t
(legal? 6 '(4 8))   #f
(legal? 8 '(4 8))   #f

(define solution? (lambda (conf) (= (length conf) fresh-try)))
```

Le problème des 8 reines IV

Le cœur de notre stratégie consiste à prolonger une configuration en une solution. Pour éviter tant les omissions que les répétitions, on doit ordonner totalement l'ensemble des configurations, de manière compatible avec l'ordre dans lequel elles seront générées et testées. L'ordre *total* le plus communément utilisé pour les domaines de listes est l'ordre lexicographique. Nous l'adoptons ici avec deux modifications mineures : il est renversé (l'élément de poids le plus élevé est à la fin) et inversé (un élément plus grand précède un élément plus petit). Notons aussi qu'une configuration partielle précédera toujours ses prolongements.

Pour fixer les idées, considérons le cas $n = 8$. A chaque configuration partielle α, nous associons le nombre dont la liste renversée des chiffres est α', obtenue en complétant α par des occurrences de 9. Par exemple, aux configurations $\alpha =$ `(7 3 1)` et $\beta =$ `(2 6 3 1)` on associe respectivement les nombres $\alpha' = 13799999$ et $\beta' = 13629999$. L'ordre des configurations est l'ordre numérique inverse ; la configuration α précède la configuration β (on écrit $\alpha \prec \beta$) puisque l'on a $\alpha' > \beta'$. La première configuration est donc la configuration vide `()`, représentée par le nombre 99999999.

Le problème des 8 reines V

Prolongement d'une configuration sans "backtracking". La fonction `build-solution` calcule le premier *prolongement* d'une configuration donnée (s'il existe).

```
(define fresh-try 8)

(define build-solution
  (lambda (conf)
    (newline) (blank-write conf)
    (if (solution? conf)
        conf
        (forward fresh-try conf))))

(define forward
  (lambda (try conf)
    (cond ((zero? try) 'none)
          ((legal? try conf) (build-solution (cons try conf)))
          (else (forward (sub1 try) conf)))))
```

Le problème des 8 reines VI

```
(build-solution '(2 6 3 1 4 8))
        (2 6 3 1 4 8)
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8)
:-) (5 7 2 6 3 1 4 8)

(build-solution '(3 1 7 2 4 8))
        (3 1 7 2 4 8)
      (5 3 1 7 2 4 8)
:-) none

(build-solution '(6 2 7 1 4 8))
        (6 2 7 1 4 8)
      (3 6 2 7 1 4 8)
:-) none   ;; induit par forward
```

Le problème des 8 reines VII

```
(define forward
  (lambda (try conf)
    (cond ((zero? try) (backtrack conf))
          ((legal? try conf) (build-solution (cons try conf)))
          (else (forward (sub1 try) conf)))))

(build-solution '(6 2 7 1 4 8))
        (6 2 7 1 4 8) ;; configurations
      (3 6 2 7 1 4 8)
      (3 6 2 7 1 4 8) ;; intermediaires
        (6 2 7 1 4 8)
          (2 7 1 4 8) ;; on pourrait
            (7 1 4 8)
            (3 1 4 8) ;; les afficher
          (6 3 1 4 8)
        (2 6 3 1 4 8) ;; en utilisant
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8) ;; newline-display
:-) (5 7 2 6 3 1 4 8)
```

Le problème des 8 reines VIII

```
(define backtrack
  (lambda (conf)
    (newline) (blank-write conf) ;; optionnel
    (if (null? conf)
        '()
        (forward (sub1 (car conf)) (cdr conf)))))

(build-solution '(6 1 4 8))
            (6 1 4 8)
          (2 6 1 4 8)
        (7 2 6 1 4 8)
        (5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
      (7 5 2 6 1 4 8)
        (5 2 6 1 4 8)
          (2 6 1 4 8)
            (6 1 4 8)
            (3 1 4 8)
          (6 3 1 4 8)
        (2 6 3 1 4 8)
      (7 2 6 3 1 4 8)
    (5 7 2 6 3 1 4 8)
:-) (5 7 2 6 3 1 4 8)
```

Le problème des 8 reines IX

Construction progressive de toutes les solutions.

```
(build-solution '()) :-) (5 7 2 6 3 1 4 8) ;; solution 01

(backtrack '(5 7 2 6 3 1 4 8)) :-) (4 7 5 2 6 1 3 8) ;; solution 02

(backtrack '(4 7 5 2 6 1 3 8)) :-) (6 4 7 1 3 5 2 8) ;; solution 03

(backtrack '(6 4 7 1 3 5 2 8)) :-) (6 3 5 7 1 4 2 8) ;; solution 04

... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...

(backtrack '(3 5 2 8 6 4 7 1)) :-) (5 2 4 7 3 8 6 1) ;; solution 91

(backtrack '(5 2 4 7 3 8 6 1)) :-) (4 2 7 3 6 8 5 1) ;; solution 92

(backtrack '(4 2 7 3 6 8 5 1)) :-) ()           ;; plus de solution
```

Il existe donc 92 solutions.

Le problème des 8 reines X

Automatiser en une boucle le “backtracking” sur les solutions.

```
(define build-all-solutions
  (lambda ()
    (letrec
      ((loop
        (lambda (sol)
          (if (null? sol)
              '()
              (cons sol (loop (backtrack sol)))))))
      (loop (build-solution '()))))))

(build-all-solutions)
:-) ((5 7 2 6 3 1 4 8) (4 7 5 2 6 1 3 8) ... (4 2 7 3 6 8 5 1))
```

Le problème des 8 reines XI

```
(define step  ;; sans récursion mutuelle
  (lambda (try conf)
    (cond ((zero? try)
           (if (null? conf) '() (step (sub1 (car conf)) (cdr conf))))
          ((legal? try conf)
           (let ((lp (cons try conf)))
             (if (solution? lp) lp (step fresh-try lp))))
          (else (step (sub1 try) conf)))))

(define build-all-solutions
  (lambda ()
    (letrec
      ((loop
        (lambda (sol)
          (if (null? sol)
              '()
              (cons sol (loop (step (sub1 (car sol)) (cdr sol))))))))
      (loop (step fresh-try '()))))))
```

Voyageur de commerce I

Données : un ensemble de villes numérotées de `1` à `n`,
une carte `dmap` des distances entre villes,
une longueur maximale de circuit `lmax`

```
0   200 316 200 316 282 447 412 423 632 141 223
    0   141 282 316 200 400 223 316 600 141 100
        0   423 447 316 510 223 400 707 282 100
            0   141 200 282 412 316 447 141 360
                0   141 141 360 200 316 200 412
                    0   200 223 141 400 141 300
                        0   360 141 200 316 500
                            0   223 538 300 282
                                0   316 282 412
                                    0   510 700
                                        0   223
                                            0
```

Voyageur de commerce II

Données : un ensemble de villes numérotées de `1` à `n`,
une carte `dmap` des distances entre villes,
une longueur maximale de circuit `lmax`

```
(define dmap
  (lambda (i j)
    (cond ((= i j) 0)
          ((> i j) (dmap j i))
          ((= i 1) (cond ((= j 2) 200)
                         ...       ...
                         ((= j 12) 316)))
          ((= i 2) ...)
          ...       ...
          ((= i 11) (cond ((= j 12) 223))))))
```

Voyageur de commerce III

Essai d'adjonction de la ville `try` au circuit partiel légal `legal-c`.
La solution `(3 2 4 1)` représente le circuit $1-3-2-4-1$.

```
(define legal?
  (lambda (try legal-c dmap lmax)
    (letrec ((circ? (lambda (t l-c)
                      (or (null? l-c)
                          (and (not (= t (car l-c)))
                               (circ? t (cdr l-c))))))
             (size-c (lambda (l-c)
                       (cond ((= (car l-c) 1) 0)
                             (else (+ (size-c (cdr l-c))
                                      (dmap (car l-c) 1)
                                      (dmap (car l-c) (cadr l-c))
                                      (- (dmap (cadr l-c) 1)))))))))
      (and (circ? try legal-c) (<= (size-c (cons try legal-c)) lmax)))))
```

Voyageur de commerce IV

```
(define fresh-try 12)

(define solution?
  (lambda (legal-c) (= (length legal-c) fresh-try)))

(define build-solution
  (lambda (legal-c dmap lmax)
    (if (solution? legal-c)
        legal-c
        (forward fresh-try legal-c dmap lmax))))

(define forward
  (lambda (try legal-c dmap lmax)
    (cond ((zero? try) (backtrack legal-c dmap lmax))
          ((legal? try legal-c dmap lmax)
           (build-solution (cons try legal-c) dmap lmax))
          (else (forward (sub1 try) legal-c dmap lmax)))))

(define backtrack
  (lambda (legal-c dmap lmax)
    (if (null? legal-c)
        '()
        (forward (sub1 (car legal-c)) (cdr legal-c) dmap lmax))))
```

Voyageur de commerce V

```
(define fresh-try 5)   fresh-try
  200 316 200 316
      141 282 316
          423 447
              141

(define length-c
  (lambda (l-c dmap)
    (if (= (car l-c) 1)
        0
        (+ (length-c (cdr l-c) dmap)
           (dmap (car l-c) 1)
           (dmap (car l-c) (cadr l-c))
           (- (dmap (cadr l-c) 1))))))
```

Voyageur de commerce VI

```
(build-solution '(1) dmap 1000)  ()
(build-solution '(1) dmap 2000)  (2 3 4 5 1)
(length-c '(2 3 4 5 1) dmap)     1221
(backtrack '(2 3 4 5 1) dmap 1221) > (3 2 4 5 1)
(length-c '(3 2 4 5 1) dmap)     1196
(backtrack '(3 2 4 5 1) dmap 1196) > (2 3 5 4 1)
(length-c '(2 3 5 4 1) dmap)     1129
(backtrack '(2 3 5 4 1) dmap 1129) > (3 2 5 4 1)
(length-c '(3 2 5 4 1) dmap)     1114
(backtrack '(3 2 5 4 1) dmap 1114) > (4 5 2 3 1)
(length-c '(4 5 2 3 1) dmap)     1114
(backtrack '(4 5 2 3 1) dmap 1114) > ()
```

Voyageur de commerce VII

```
(define fresh-try 8)   fresh-try

  200 316 200 316 282 447 412
      141 282 316 200 400 223
          423 447 316 510 223
              141 200 282 412
                  141 141 360
                      200 223
                          360

(build-solution '(1) dmap 1500)              (2 3 8 6 7 5 4 1)

(length-c '(2 3 8 6 7 5 4 1) dmap)           1469

(backtrack '(2 3 8 6 7 5 4 1) dmap 1469)     (4 5 7 6 8 3 2 1)

(length-c '(4 5 7 6 8 3 2 1) dmap)           1469

(backtrack '(4 5 7 6 8 3 2 1) dmap 1469)     ()
```

Recherche abstraite I

Idée.

On observe que le problème des reines et celui du voyageur de commerce, quoique très différents en apparence, se résolvent par la même technique.

La constante `fresh-try`, ainsi que les procédures `legal?` et `solution?` diffèrent d'un problème de recherche à l'autre ; ces objets seront donc des arguments de la procédure de recherche abstraite `searcher`.

Par contre, le principe de construction des solutions ne varie pas, donc les procédures `build-solution`, `forward`, `backtrack` et `build-all-solutions` seront écrites une fois pour toutes. Elles pourront être internes à la procédure de recherche abstraite `searcher`.

L'unification de plusieurs problèmes concrets en un problème plus abstrait peut impliquer quelques changements de détail.

Recherche abstraite II

```
(define searcher
  (lambda (legal? solution? fresh-try)
    (letrec
      ((build-solution
         (lambda (conf)
           (if (solution? conf) conf (forward fresh-try conf))))
       (forward
         (lambda (try conf)
           (cond ((zero? try) (backtrack conf))
                 ((legal? try conf) (build-solution (cons try conf)))
                 (else (forward (sub1 try) conf)))))
       (backtrack
         (lambda (conf)
           (if (null? conf) '() (forward (sub1 (car conf)) (cdr conf)))))
       (build-all-solutions
         (lambda ()
           (letrec
             ((loop
                (lambda (sol)
                  (if (null? sol) '() (cons sol (loop (backtrack sol)))))))
             (loop (build-solution '())))))))
      (build-all-solutions))))
```

Recherche abstraite III

```
(define legal?-reines
  (lambda (try conf)
    (letrec
      ((good?
         (lambda (new-pl up down)
           (cond ((null? new-pl) #t)
                 (else (let ((next-pos (car new-pl)))
                         (and (not (= next-pos try))
                              (not (= next-pos up))
                              (not (= next-pos down))
                              (good? (cdr new-pl)
                                     (add1 up)
                                     (sub1 down)))))))))
      (good? conf (add1 try) (sub1 try)))))

(define solution?-reines
  (lambda (s) (= (length s) fresh-try-reines)))

(define fresh-try-reines 8)
```

Problème des 8 reines : essais

```
(searcher legal?-reines solution?-reines fresh-try-reines)
:-) ((5 7 2 6 3 1 4 8) ... (4 2 7 3 6 8 5 1))

(length (searcher legal?-reines solution?-reines fresh-try-reines))
:-) 92

(define fresh-try-reines 6)
:-) ...

(searcher legal?-reines solution?-reines fresh-try-reines)
:-) ((2 4 6 1 3 5) (3 6 2 5 1 4) (4 1 5 2 6 3) (5 3 1 6 4 2))

(define fresh-try-reines 10)
:-) ...

(length (searcher legal?-reines solution?-reines fresh-try-reines))
:-) 724
```

Paramètres pour le problème du voyageur

```
(define dmap
  (lambda (i j)
    (cond ((= i j) 0)
          ((> i j) (dmap j i))
          ((= i 0) (cond ((= j 1) 200)
                         ((= j 2) 316))
                         ...  ...  ...
                         ((= j 10) 141))
                         ((= j 11) 223)))
          ((= i 1) (cond ((= j 2) 141))
                         ...  ...  ...
          ((= i 2) (cond ((= j 3) 423))
          ... ... ... ... ... ... ...
          ((= i 9) (cond ((= j 10) 510))
                         ((= j 11) 700)))
          ((= i 10)(cond ((= j 11) 223))))))

(define lmax ...)

(define fresh-try-voyageur ...)
```

Procédures pour le problème du voyageur

```
(define legal?-voyageur
  (lambda (try leg-c)
    (letrec
      ((circ?
         (lambda (t l-c)
           (cond ((null? l-c) #t)
                 (else (and (not (= t (car l-c))) (circ? t (cdr l-c)))))))
       (size-c
         (lambda (l-c)
           (cond ((null? l-c) 0)
                 ((null? (cdr l-c)) (* 2 (dmap (car l-c) 0)))
                 (else (+ (size-c (cdr l-c))
                          (dmap (car l-c) 0)
                          (dmap (car l-c) (cadr l-c))
                          (- (dmap (cadr l-c) 0)))))))))
      (and (circ? try leg-c) (<= (size-c (cons try leg-c)) lmax)))))

(define solution?-voyageur (lambda (s) (= (length s) fresh-try-voyageur)))
```

Problème du voyageur : essais

```
; Villes de 0 a 4, longueur maximale 1200
(define lmax 1200) ...
(define fresh-try-voyageur 4) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((2 1 3 4) (1 2 4 3) (2 1 4 3) (3 4 1 2) (4 3 1 2) (3 4 2 1))
; Rappel: (2 1 3 4) est 0-2-1-3-4-0


; longueur maximale 1150
(define lmax 1150) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 4 3) (2 1 4 3) (3 4 1 2) (3 4 2 1))
```

Problème du voyageur : essais (bis)

```
; Villes de 0 a 8 (longueur maximale 1150)
(define fresh-try-voyageur 8) ...

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ()
; pas de solution


; (Villes de 0 a 8) longueur maximale 1600
(define lmax 1600) lmax

(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 5 8 6 4 3) (3 4 6 8 5 7 2 1))
; circuit 0-1-2-7-5-8-6-4-3-0 ou inverse
```

Problème du voyageur : essais (ter)

```
; Villes de 0 a 9, longueur maximale 2000
(define fresh-try-voyageur 9) (define lmax 2000)


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 8 6 9 4 5 3) (1 2 7 8 9 6 4 5 3)
     (1 2 7 5 8 6 9 4 3) (1 2 7 8 5 6 9 4 3)
     (1 2 7 5 8 9 6 4 3) (1 2 7 8 9 6 5 4 3)
     (3 4 5 6 9 8 7 2 1) (3 5 4 6 9 8 7 2 1)
     (3 5 4 9 6 8 7 2 1) (3 4 9 6 5 8 7 2 1)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))


(define lmax 1925) ; longueur maximale 1925


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ()


(define lmax 1926) ; longueur maximale 1926


(searcher legal?-voyageur solution?-voyageur fresh-try-voyageur)
:-) ((1 2 7 5 8 6 9 4 3) (1 2 7 5 8 9 6 4 3)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))
```

Variables globales

La valeur de `(searcher ...)` dépend des variables globales `dmap` et `lmax` ; il vaut mieux éliminer ce manque de transparence.

"Solution" naïve et **incorrecte**

```
(let ((lmax 3000))
     (searcher legal?-voyageur solution?-voyageur fresh-try-voyageur))
:-) ((1 2 7 5 8 6 9 4 3) (1 2 7 5 8 9 6 4 3)
     (3 4 6 9 8 5 7 2 1) (3 4 9 6 8 5 7 2 1))
```

Le `let` n'a eu aucun effet ;
la réponse correspond à la valeur globale de `lmax`.

Pour l'évaluateur, l'environnement pertinent pour la variable `lmax` est l'environnement global, où la variable `legal?-voyageur` a été LIEE, et non l'environnement d'APPLICATION, où la forme `(let ...)` a été évaluée.

Rappelons l'approche *lexicale* de Scheme en ce qui concerne la portée des identificateurs. L'identificateur `lmax` n'ayant aucune occurrence dans le *texte* du corps du `let`, il est normal que la liaison due au `let` soit restée sans effet.

Suppression des variables globales

Eviter les variables globales `dmap` et `lmax`

Solution correcte

On n'écrit pas directement la procédure `legal?-voyageur` mais le générateur de procédure `make-legal?-voyageur`, tel que la forme `(make-legal?-voyageur map lmax)` ait la valeur (prédicative) du `legal?-voyageur` antérieur.

On pourra alors définir

```
(define solve-voyageur
  (lambda (dmap lmax)
    (searcher (make-legal?-voyageur dmap lmax)
              solution?-voyageur
              fresh-try-voyageur)))
```

Il est préférable d'inclure les définitions de `make-legal?-voyageur`, `solution?-voyageur` et `fresh-try-voyageur` dans `solve-voyageur`.

```
(define make-legal?-voyageur
  (lambda (dmap lmax)
    (lambda (try leg-c)
      (letrec
        ((circ?
           (lambda (try leg-c)
             (cond ((null? leg-c) #t)
                   (else (and (not (= try (car leg-c)))
                              (circ? try (cdr leg-c)))))))
         (size-c
           (lambda (l-c)
             (cond ((null? l-c) 0)
                   ((null? (cdr l-c)) (* 2 (dmap (car l-c) 0)))
                   (else (+ (size-c (cdr l-c))
                            (dmap (car l-c) 0)
                            (dmap (car l-c) (cadr l-c))
                            (- (dmap (cadr l-c) 0)))))))))
        (and (circ? try leg-c)
             (<= (size-c (cons try leg-c)) lmax))))))
```

14. Les vecteurs

Comment regrouper des objets en une structure ?

```
(list 4 'x 3 '(7 . a))                (4 x 3 (7 . a))
(vector 4 'x 3 '(7 . a))             #(4 x 3 (7 . a))
```

L'accès aux éléments d'une liste est séquentiel ;
l'accès aux éléments d'un vecteur est direct

```
(vector-length '#(4 x 3 (7 . a)))   4
(vector-ref '#(4 x 3 (7 . a)) 0)    4
(vector-ref '#(4 x 3 (7 . a)) 3)    (7 . a)
(vector-ref '#(4 x 3 (7 . a)) 4)    Object 4 out of range
(subvector '#(4 x 3 (7 . a)) 1 2)   #(x)
(vector->list '#(4 x 3 (7 . a)))    (4 x 3 (7 . a))
(list->vector '(4 x 3 (7 . a)))     #(4 x 3 (7 . a))
```

Instructions altérantes

On utilise les vecteurs plutôt que les listes si l'accès aléatoire est important.

En scheme pur, on n'altère pas les objets, on en crée une copie modifiée.

C'est plus lent et cela prend de la place ... scheme comporte aussi des instructions altérantes.

```
(define v '#(0 2 4 6 8 10 12 14 16 18))    v
(define w v)                               w
(vector-set! v 3 -99)                      No value
v                                          #(0 2 4 -99 8 10 12 14 16 18)
w                                          #(0 2 4 -99 8 10 12 14 16 18)
```

C'est dangereux (effets de bord) ... mais c'est utile !

Remarque. On peut aussi altérer des variables simples, des listes, etc. au moyen de `set!`, `set-car!`, `set-cdr!`.

Tri par insertion (rappel)

```
(define insertsort
  (lambda (ls)
    (if (null? ls)
        ls
        (insert (car ls) (insertsort (cdr ls))))))

(define insert
  (lambda (a ls)
    (cond ((null? ls) (cons a '()))
          ((< a (car ls)) (cons a ls))
          (else (cons (car ls) (insert a (cdr ls)))))))
```

Version altérante pour vecteurs

```
(define vector-insertsort!
  (lambda (v)
    (let ((size (vector-length v)))
      (letrec ((loop
                (lambda (k)
                  (if (< k size)
                      (begin (vector-insert! k v) (loop (+ k 1)))))))
        (loop 1)))))

(define vector-insert!
  (lambda (k vec)
    (let ((val (vector-ref vec k)))
      (letrec ((insert-h
                (lambda (m)
                  (if (zero? m)
                      (vector-set! vec 0 val)
                      (let ((c (vector-ref vec (- m 1))))
                        (if (< val c)
                            (begin (vector-set! vec m c) (insert-h (- m 1)))
                            (vector-set! vec m val)))))))
        (insert-h k)))))
```

Documentation et essais I

Spécification. Si `v` est (lié à) un vecteur dans l'environnement courant avant l'exécution de `(vector-insertsort! v)`, alors `v` est (lié à) la version triée de ce vecteur après l'exécution.

Fonctionnement. Si s est la taille de `v`, exécuter `(vector-insertsort! v)` revient à exécuter la séquence

```
(vector-insert! 1 v)
(vector-insert! 2 v)
⋮       ⋮       ⋮       ⋮
(vector-insert! n v)
```

où $n = s-1$ est l'index du dernier élément du vecteur `v`.
Rappel : un vecteur de taille s est indexé de 0 à $s-1$.

Spécification. Si le préfixe `v`$[0:k-1]$ est trié avant l'exécution de la forme `(vector-insert!` k `v)`, alors cette exécution a pour effet d'insérer `v`$[k]$ à sa place. Le suffixe `v`$[k+1:n]$ n'est pas altéré.

Documentation et essais II

Exemple. Pour un vecteur de longueur 6, on appelle 5 fois la procédure d'insertion.

```
(define v '#(9 3 7 0 5 1))        #(9 3 7 0 5 1)
                                    ^ ^

(vector-insert! 1 v)              #(3 9 7 0 5 1)
                                      ^ ^

(vector-insert! 2 v)              #(3 7 9 0 5 1)
                                        ^ ^

(vector-insert! 3 v)              #(0 3 7 9 5 1)
                                          ^ ^

(vector-insert! 4 v)              #(0 3 5 7 9 1)
                                            ^ ^

(vector-insert! 5 v)              #(0 1 3 5 7 9)
```

Documentation et essais III

Fonctionnement. L'appel de `(vector-insert! k v)` crée la liaison `val: v[k]` puis provoque le nombre adéquat de "décalages", suivi de l'écrasement de la dernière case recopiée par `val`.

```
...   ...
  ;; (insert-h m) ::=
  (let ((c (vector-ref vec (- m 1))))
    (if (< val c)
        (begin (vector-set! vec m c) (insert-h (- m 1)))
        (vector-set! vec m val)))
...   ...
```

```
k: 4   v: #(0 3 7 9 5 1))    val: 5
init   v: #(0 3 7 9 5 1))    val: 5    c: 9
then   v: #(0 3 7 9 9 1))    val: 5    c: 7
then   v: #(0 3 7 7 9 1))    val: 5    c: 3
else   v: #(0 3 5 7 9 1))
```

“Reverse”

“Reverse”, version fonctionnelle simple

```
(define reverse
  (lambda (l)
    (if (null? l)
        '()
        (append (reverse (cdr l)) (list (car l))))))
```

“Reverse”, version fonctionnelle accumulante

```
(define rev-it
  (lambda (l)
    (letrec ((r0 (lambda (u v)
                   (if (null? u)
                       v
                       (r0 (cdr u) (cons (car u) v))))))
      (r0 l '()))))
```

Version vectorielle altérante

```
(define vector-reverse!
  (lambda (v)
    (let ((s (vector-length v)) (t 0))
       (letrec
         ((switch (lambda (i j)
                    (begin (set! t (vector-ref v i))
                           (vector-set! v i
                           (vector-ref v j))
                           (vector-set! v j t))))
          (loop (lambda (i j)
                  (if (< i j)
                      (begin (switch i j)
                             (loop (+ i 1) (- j 1)))))))
         (loop 0 (- s 1))))))
```

Générateur aléatoire

```
(make-vector 4)   #(() () () ())

(random 1000)   624    ;; [0...999]

(define random-vector
  (lambda (n)
    (let ((v (make-vector n)))
      (letrec ((fill
                (lambda (i)
                  (if (< i n)
                      (begin (vector-set! v i (random 1000))
                             (fill (+ i 1)))))))
        (fill 0))
      v)))

(random-vector 5)    #(90 933 656 240 587)
(random-vector 5)    #(666 943 203 632 512)
```

15. Mécanismes particuliers, tabulation

Mesure du temps d'exécution I

Définition d'un "chronomètre".

```
(define time
  (lambda (proc arg) ;; arg : argument unique
    (let* ((t0 (runtime))
           (val (proc arg))
           (t1 (runtime))
           (del (round (* (- t1 t0) 100))))
      (newline) (display "Time = ") (write del)
      val)))
```

Fonctions de test.

```
(define fib (lambda (n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))))

(define fib2
  (lambda (n)
    (letrec ((f (lambda (n a b) (if (= n 0) a (f (- n 1) b (+ a b))))))
      (f n 0 1))))
```

Mesure du temps d'exécution II

Essais.

```
(time fib 20)   Time = 455.   ;Value:   6765
(time fib 21)   Time = 735.   ;Value:  10946

(time fib2 20)  Time =   0.   ;Value:   6765
(time fib2 30)  Time =   0.   ;Value: 832040
```

	300	1000	3000	10000
reverse	20	170	1435	15428
rev-it	5	15	43	140
vector-reverse!	8	27	72	237

```
(eval '(+ 5 1 3) '())   9
(apply + '(5 1 3))      9
```

Fonctions d'arité variable

```
(define list (lambda v v))
(list 1 2 3 4)                (1 2 3 4)

(define sort (lambda args (apply insertsort (list args))))

(sort)                        ()
(sort 3 1 5 2)                (1 2 3 5)

(define g-expt
  (lambda (x . l) (if (null? l) x (expt x (apply g-expt l)))))

(g-expt)                      Error: at least 1 argument required
(g-expt 5)                    5
(g-expt 2 3)                  8
(g-expt 2 3 2)                512
```

Mesure du temps d'exécution III

```
(define time*
  (lambda (proc . args) ;; args: liste d'arguments
    (let* ((t0 (runtime))
           (val (apply proc args))
           (t1 (runtime))
           (del (round (* (- t1 t0) 100))))
      (newline) (display "Time = ") (write del)
      val)))
```

```
(time* fib 20)
 Time = 457.   ;Value: 6765

(time* + (fib 18) (fib 19))
 Time =   0.   ;Value: 6765
```

Seul le temps consommé par l'addition a été comptabilisé !

```
(time* g-expt 2 2 2 2)     Time =   0.    65536
(time* g-expt 2 2 2 2 2)   Time = 868.    **too big!**
```

Simulation de `letrec`

```
(letrec
  ((even? (lambda (n) (or  (= n 0) (odd?  (- n 1)))))
   (odd?  (lambda (n) (and (> n 0) (even? (- n 1))))))
  (list (even? 11) (odd? 7)))
;Value: (#f #t)

(let ((even? 'any) (odd?  'any))
  (let ((e (lambda (n) (or  (= n 0) (odd?  (- n 1)))))
        (o (lambda (n) (and (> n 0) (even? (- n 1))))))
    (set! even? e)
    (set! odd?  o)
    (list (even? 11) (odd? 7))))
;Value: (#f #t)
```

Une curiosité . . .

On peut se passer de `letrec`. Le code

```
(letrec
  ((fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))))
  (fact x))
```

peut être simulé par le code

```
((lambda (n)
   ((lambda (fact) (fact fact n))
    (lambda (f k) (if (= k 0) 1 (* k (f f (- k 1)))))))
 x)
```

Tables et tabulation I

Idée : ne pas recalculer un élément déjà calculé.

Exemple : rendre efficace le programme naïf `fib`.

Stratégie : maintenir une table de valeurs calculées.

Une table est une liste de paires pointées (variable,valeur)

```
(define *TABLE-fib* '((1 . 1) (0 . 0)))

(define m-fib
  (lambda (n)
    (let ((v (assv n *TABLE-fib*)))
      (if v
          (cdr v)
          (let ((a (m-fib (- n 1))) (b (m-fib (- n 2))))
            (let ((val (+ a b)))
              (set! *TABLE-fib* (cons (cons n val) *TABLE-fib*))
              val))))))
```

Tables et tabulation II

Essais.

```
(time m-fib 100)   Time = 15.          354224848179261915075

(time fib-it 100)  Time = 2.           354224848179261915075

(time m-fib 100)   Time = 0.           354224848179261915075

(time m-fib 90)    Time = 0.           2880067194370816120
```

Problèmes.

Version tabulée à écrire pour chaque fonction.

Introduction d'une variable globale.

Tables et tabulation III

Fonctions de tabulation.

```
(define lookup
  (lambda (obj table succ-p fail-p)
    (if (null? table)
        (fail-p)
        (let ((pr (car table)))
          (if (equal? (car pr) obj)
              (succ-p pr)
              (lookup obj (cdr table) succ-p fail-p))))))

(define assoc
  (lambda (obj table)
    (lookup obj table (lambda (pr) pr) (lambda () #f))))
```

Tables et tabulation IV

Un tabulateur générique.

```
(define memoize
  (lambda (proc)
    (let ((table '()))
      (lambda (arg)
        (lookup
          arg
          table
          cdr
          (lambda ()
            (let ((val (proc arg)))
              (set! table (cons (cons arg val) table))
              val)))))))
```

Tables et tabulation V

Fonctions de test.

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))

(define m-fib1 (memoize fib))

(define m-fib2
  (memoize (lambda (n)
             (if (< n 2)
                 n
                 (+ (m-fib2 (- n 1)) (m-fib2 (- n 2)))))))

(define fib-it
  (lambda (n)
    (letrec
      ((f0 (lambda (n p q)
             (if (= n 0) p (f0 (- n 1) q (+ p q))))))
      (f0 n 0 1))))
```

Tables et tabulation VI

Essais.

```
(time fib 20)     Time: 161. ;Value:  6765
(time fib 20)     Time: 161. ;Value:  6765
(time fib 21)     Time: 268. ;Value: 10946

(time m-fib1 20)  Time: 165. ;Value:  6765
(time m-fib1 21)  Time: 268. ;Value: 10946
(time m-fib1 20)  Time:   0. ;Value:  6765
(time m-fib1 19)  Time: 100. ;Value:  4181
(time m-fib1 21)  Time:   0. ;Value: 10946

(time m-fib2 20)  Time:   2.
(time m-fib2 20)  Time:   0.
(time m-fib2 22)  Time:   0. ;Value: 17711
(time m-fib2 100) Time:  29. ;Value: ...
(time m-fib2 100) Time:   0. ;Value: ...

(time fib-it 100) Time:   1. ;Value: ...
(time fib-it 100) Time:   1. ;Value: ...
```

```
(define num-part
  (lambda (n k)
    (if (or (= k n) (= k 1))
        1
        (+ (num-part (- n 1) (- k 1))
           (* k (num-part (- n 1) k))))))

(time* num-part 10  5)   Time = 0.01
(time* num-part 20 10)   Time = 4.87
```

indice	0	1	2	3	4	5	6	...
(n,k)	(1,1)	(2,1)	(2,2)	(3,1)	(3,2)	(3,3)	(4,1)	...

La fonction

$$(n,k) \;\mapsto\; \frac{n(n-1)}{2} + k - 1$$

calcule l'indice associé à des arguments donnés.

```
(define index
  (lambda (n k)
    (+ (/ (* n (- n 1)) 2) k -1)))

(define *MAX* 300)    ;; valeur maximale de n

(define *MEMO* (make-vector (index *MAX* *MAX*) #f))

(define num-part-glob
  (lambda (n k)
    (let ((w (vector-ref *MEMO* (index n k))))
      (if w
          w
          (let
            ((x
              (if (or (= k n) (= k 1))
                  1
                  (+ (num-part-glob (- n 1) (- k 1))
                     (* k
                        (num-part-glob (- n 1) k))))))
            (vector-set! *MEMO* (index n k) x)
            x)))))
```

Si on refuse l'usage des variables globales, on peut écrire

```
(define num-part-loc
  (lambda (n k)
    (let ((table       ;; table locale
           (make-vector (index n n) #f)))
      (letrec
          ((aux
            (lambda (m q)
              (let ((w (vector-ref table (index m q))))
                (if w
                    w
                    (let
                      ((x
                        (if (or (= q 1) (= q m))
                            1
                            (+ (aux (- m 1) (- q 1))
                               (* q (aux (- m 1) q))))))
                      (vector-set! table (index m q) x)
                      x))))))
        (aux n k)))))
```

Il est intéressant d'observer la manière dont la table se remplit en cours d'exécution ; cela peut se faire en intercalant dans le code un ordre d'impression après chaque modification de la table :

```
(num-part-loc 5 3)  ==>
   #(#f #f #f #f #f  1 #f #f #f #f #f #f #f #f)
   #(#f #f  1 #f #f  1 #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f #f  1 #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f  3  1 #f #f #f #f #f #f #f #f)
   #(#f  1  1 #f  3  1 #f #f  6 #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f #f  6 #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f  7  6 #f #f #f #f #f)
   #(#f  1  1  1  3  1 #f  7  6 #f #f #f 25 #f)
25
```

L'inconvénient d'une table locale est que chaque appel à `num-part-loc` provoque la création d'une nouvelle table. On peut éviter cet inconvénient en utilisant une variable libre :

```
(define num-part-free
  (let ((table (make-vector (index *MAX* *MAX*) #f)))
    (letrec
        ((aux
          (lambda (m q)
            (let ((w
                   (vector-ref table (index m q))))
              (if w
                  w
                  (let
                    ((x
                      (if (or (= q 1) (= q m))
                          1
                          (+ (aux (- m 1) (- q 1))
                             (* q (aux (- m 1) q))))))
                    (vector-set! table (index m q) x)
                    x))))))
      aux)))
```

La table est créée lors de l'évaluation de la forme `define` et réutilisée à chaque appel, comme le montre la session suivante :

```
(timer* num-part-loc 200 100)   Time = 1.73
(timer* num-part-loc 200 100)   Time = 1.73
(timer* num-part-free 200 100)  Time = 1.73
(timer* num-part-free 200 100)  Time = 0.00
```

Avec la table locale, deux calculs successifs de la même valeur (non affichée ici car il s'agit d'un nombre de 235 chiffres) prennent le même temps ; avec la table non locale, le second "calcul" est instantané, car la valeur est trouvée dans la table.

```
(define *cons* 0)

(define kons
  (lambda (x y) (set! *cons* (1+ *cons*)) (cons x y)))

(define lpref1
  (lambda (l)
    (if (null? l)
        (kons l '())
        (kons '() (put (car l) (lpref1 (cdr l)))))))

(define put
  (lambda (x ll)
    (if (null? ll)
        '()
        (kons (kons x (car ll)) (put x (cdr ll))))))
```

```
(define lpref2
  (lambda (l)
    (if (null? l)
        (kons l '())
        (kons l (lpref2 (butlast l))))))

(define butlast
  (lambda (l)
    (if (null? (cdr l))
        '()
        (kons (car l) (butlast (cdr l))))))
```

```
(define lpref3
  (lambda (l) (lreverse (lsuff (reverse l)))))

(define reverse (lambda (l) (rev l '())))

(define lsuff
  (lambda (l)
    (if (null? l) (kons l '()) (kons l (lsuff (cdr l))))))

(define lreverse
  (lambda (ll)
    (if (null? ll)
        '()
        (kons (rev (car ll) '()) (lreverse (cdr ll))))))

(define rev
  (lambda (l a)
    (if (null? l)
        a
        (rev (cdr l) (kons (car l) a)))))
```

```
(define count-1 (lambda (n) (set! *cons* 0) (lpref1 (enum 1 n)) *cons*))
(define count-2 (lambda (n) (set! *cons* 0) (lpref2 (enum 1 n)) *cons*))
(define count-3 (lambda (n) (set! *cons* 0) (lpref3 (enum 1 n)) *cons*))

(define enum (lambda (p q) (if (> p q) '() (cons p (enum (+ p 1) q)))))

(define *list-length* '(1 2 5 10 50 100))
```

`*list-length*`	(1 2 5 10 50 100)
`(map count-1 *list-length*)`	(4 9 36 121 2601 10201)
`(map count-2 *list-length*)`	(2 4 16 56 1276 5051)
`(map count-3 *list-length*)`	(6 11 32 87 1427 5352)