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))))))
```


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 ?

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
      3.9999999999999036    ;; précision excellente
      5.000000000000184     ;; précision excellente
      6.000000000000003)    ;; précision excellente
```


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
```

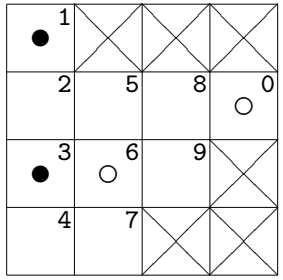
On calcule de même quel taux de chargement correspond à un taux d'intérêt mensuel de 0.5 % :

```
(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`.

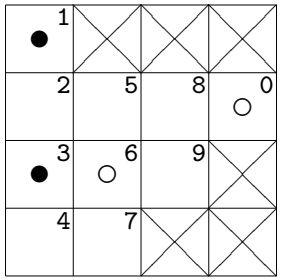
Mouvement `(a . b)` : de la position `a` vers la position `b`.

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)
```