La lettre de Caml numéro 2

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. (1) 48 05 16 04
Fax (1) 48 07 80 18
email : `cheno@micronet.fr`

novembre 1995

Édito

Dans ce numéro, qui a pris un peu de retard (mais mes copies s'accumulent…), vous trouverez un texte de notre gourou commun, Pierre Weis, qui nous parle avec le talent que nous lui connaissons d'évaluation paresseuse, nous donne une implémentation en Caml, et des applications édifiantes.
Ensuite vous trouverez quelques exemples simples de programmation en Caml, beaucoup moins ambitieux que ce que nous expose P. Weis, qui pourraient servir d'exercices dans nos classes : on s'intéresse particulièrement à un peu de combinatoire sur des mots sur $\{-1, 1\}$ *ou* $\{0, 1\}$*, et aux permutations de* $\{1, 2, \dots, n\}$*.*
Rien de génial, mais une mine d'exercices possibles.
La distribution de cette lettre est en défaut, je le sais bien, et j'essaie d'y remédier. Mais les cours, les copies, et maintenant les grèves de transport, tout cela me fait trop souvent reporter au lendemain ce que j'aurais aimé achever plus vite.
Raison de plus pour espérer plus nombreuses vos idées d'articles, même courts, et pas forcément super-géniaux…

Table des matières

Un article de Pierre Weis sur l'évaluation paresseuse

Programmation paresseuse en Caml

La programmation paresseuse consiste à différer l'évaluation de certaines parties d'un programme jusqu'à ce que cette évaluation soit absolument nécessaire à la poursuite des calculs. Cela permet de procéder à des calculs incrémentaux, et autorise la définition de structures de données potentiellement infinies. Je donne un codage simple de l'évaluation paresseuse en Caml, et une application frappante : la définition et la manipulation de séries entières en Caml.
En Caml, le régime d'évaluation est l'*appel par valeur* c'est-à-dire que l'évaluation est effectuée sans retard (on dit aussi que le langage est *strict*). Les autres régimes d'évaluation classiques des langages de programmation sont l'*appel par nom* et l'*appel par nécessité*. Tous deux consistent à retarder l'évaluation des arguments des fonctions : on appelle le code de la fonction sans avoir évalué ses arguments. Ces arguments seront évalués dans le corps de la fonction, si leur valeur est indispensable au calcul du résultat de la fonction. Par exemple si l'argument fonction n'est pas (resp. pas toujours) utilisé dans le corps de la fonction, il ne sera jamais calculé (resp. calculé que s'il est utilisé). Techniquement l'argument non évalué est appelé *glaçon* ou *suspension* ; il est passé à la fonction qui l'évalue au besoin en *forçant* (ou *dégelant*) le glaçon. La différence entre ces deux modes d'évaluation *paresseuse* concerne le nombre d'évaluations possibles des glaçons : en évaluation par nom, les glaçons sont dégelés (donc réévalués) chaque fois qu'on a besoin de leur valeur ; au contraire, avec l'évaluation par nécessité on ne calcule la valeur d'un glaçon qu'une fois au plus : on profite du dégel du glaçon pour enregistrer la valeur calculée ; ensuite toute nouvelle tentative de dégel du glaçon ne fait que retourner cette valeur enregistrée. D'un point de vue efficacité, l'appel par nécessité semble clairement gagnant : c'est en général vrai, sauf si le stockage des valeurs calculées se révèle excessivement coûteux en mémoire.
Nous allons implémenter l'appel par nécessité, en commençant par définir le type polymorphe des glaçons : un glaçon encore gelé est représenté par une fonction à zéro argument, qui donnera sa valeur quand on la déclenchera.

```
type 'a glaçon =
| Gelé of unit -> 'a
| Connu of 'a;;
Type glaçon defined.
```

Nous sommes maintenant armés pour définir les listes paresseuses en queue : la queue de ces listes est évaluée paresseusement.

```
type 'a lazy_list =
| Nil
| Cons of 'a cellule

and 'a cellule = { hd : 'a; mutable tl : 'a lazy_list glaçon};;
```

La gestion de l'appel par nécessité est faite par une fonction annexe qui dégèle les glaçons, calcule leur valeur et la stocke en mémoire :

```
let force cellule =
  let glaçon = cellule.tl in
  match glaçon with
  | Connu val -> val
  | Gelé g ->
     let val = g () in
     cellule.tl <- Connu val;
     val;;
```

Nous définissons une fonctionnelle habituelle sur les listes : un `map` paresseux qui suspend le calcul sur la queue de la liste.

```
let rec lazy_map f = function
| Nil -> Nil
| Cons ({hd = x; _} as cellule) ->
   Cons {hd = f x; tl = Gelé (function () -> lazy_map f (force cellule))};;
lazy_map : ('a -> 'b) -> 'a lazy_list -> 'b lazy_list = <fun>
```

Définir la liste potentiellement infinie des nombres entiers consiste maintenant à donner le premier élément de cette suite, puis le moyen d'en calculer le reste : il suffit pour cela d'appliquer la fonction successeur aux éléments déjà calculés …

```
let rec nat = Cons {hd = 0; tl = Gelé (fun () -> lazy_map succ nat)};;
nat : int lazy_list = Cons {hd=0; tl=Gelé <fun>}
```

Pour parcourir ces suites potentiellement infinies (par exemple en vue de les imprimer) il faut définir une fonctionnelle analogue à `do_list` ; une précaution cependant, sous peine de boucler, cette fonctionnelle ne doit pas tenter de parcourir systématiquement toute la liste ; c'est pourquoi on lui donne un argument numérique supplémentaire qui arrête arbitrairement le calcul au bout d'un certain nombres d'éléments visités.

```
let rec lazy_do_list f n = function
| Nil -> ()
| Cons ({hd = x; _} as cellule) ->
   if n > 0 then begin f x; lazy_do_list f (n - 1) (force cellule) end;;
lazy_do_list : ('a -> 'b) -> int -> 'a lazy_list -> unit = <fun>
```

Notez bien l'effet mémoire impliqué par l'impression des premiers éléments de la liste `nat` : les glaçons ont été dégelés à la demande jusqu'à la profondeur voulue.

```
lazy_do_list print_int 3 nat;;
012- : unit = ()
nat;;
- : int lazy_list =
```

```
Cons
 {hd=0;
  tl=
   Connu
    (Cons
      {hd=1;
       tl=
        Connu
         (Cons
           {hd=2;
            tl=
             Connu
              (Cons
                {hd=3;
                 tl=Gelé <fun>})})})}
```

Illustration de la programmation paresseuse : les séries entières

On peut tirer avantage de ces listes paresseuse pour implémenter des objets mathématiques eux aussi potentiellement infinis, donc *a priori* hors d'atteinte d'un calcul normal. Je vous propose de le faire pour les séries entières.
Pour cela nous utiliserons les grands nombres de Caml (la bibliothèque camlnum, qui implémente les nombres rationnels au sens des mathématiques, c'est-à-dire sans erreurs de calculs). Sous Unix, on accède à cette bibliothèque par la commande

```
$camllight camlnum
```

Commençons donc par ouvrir le module des grand nombres, et par définir un imprimeur pour ce nouveau type. Nous le définissons comme l'imprimeur par défaut des valeurs du type `num` en le déclarant au système interactif par la directive `install_printer`.

```
(* Utilitaires sur les grands nombres *)
#open "num";;

(* Impression *)
#open "format";;
let print_num n = print_string (string_of_num n);;

install_printer "print_num";;
```

Pour simplifier l'écriture nous redéfinissons les symboles arithmétiques habituels pour qu'ils opèrent sur les `num`

```
let prefix + = prefix +/
and prefix - = prefix -/
and prefix * = prefix */
```

```
and prefix / = prefix //
and prefix >= = prefix >=/
and prefix = = prefix =/;;

(* Quelques constantes *)
let un = num_of_int 1;;
let zéro = num_of_int 0;;
let moins_un = zéro - un;;
```

Le type des séries entières (potentiellement infinies) est analogue à celui des cellules de listes paresseuses, et réutilise le type des glaçons.

```
type 'a glaçon =
| Gelé of unit -> 'a
| Connu of 'a;;
type série = {Constante : num; mutable Reste : série glaçon};;
```

On définit donc de même l'accès au reste d'une série avec dégel et mise à jour pour le partage des calculs.

```
let reste_de_série s =
  match s.Reste with
  | Connu rest -> rest
  | Gelé r -> let rest = r () in s.Reste <- Connu rest; rest;;
```

La suspension automatique des calculs étant gérée, il n'y a aucune difficulté à définir les opérations usuelles sur les séries entières :

```
let rec add_série s1 s2 =
 {Constante = s1.Constante + s2.Constante;
  Reste =
   Gelé
    (function () -> add_série (reste_de_série s1) (reste_de_série s2))};;

let rec mult_série_par_constante c s =
 {Constante = c * s.Constante;
  Reste = Gelé
          (function () -> mult_série_par_constante c (reste_de_série s))};;

let rec mult_série s1 s2 =
 {Constante = s1.Constante * s2.Constante;
  Reste =
   Gelé
   (function () ->
     add_série (mult_série_par_constante s1.Constante (reste_de_série s2))
               (mult_série (reste_de_série s1) s2))};;

let opposée_de_série s = mult_série_par_constante moins_un s;;
```

L'intégration des séries ne pose pas plus de problèmes :

```
let rec integre_série c0 s =
 {Constante = c0;
  Reste = Gelé (function () -> integre_depuis un s)}

and integre_depuis n s =
 {Constante = s.Constante / n;
  Reste = Gelé (function () -> integre_depuis (n + un) (reste_de_série s))};;
```

Un petit utilitaire d'impression (naïve) des séries entières :

```
let print_variable = function
    0 -> false
  | 1 -> print_string "z"; true
  | n -> print_string "z^"; print_int n; true;;
let print_terme plus degré s =
  let c = s.Constante in
  if c = zéro then false else
  if c = un then begin print_string plus; print_variable degré end else
  if c = moins_un
   then begin print_string "- "; print_variable degré end
   else
    begin
     if c >= zéro then print_string plus else print_string "- ";
     print_num (abs_num c);
     print_variable degré
    end;;
let print_first_terme s =
  let c = s.Constante in
  if c = zéro then false else begin print_num c; true end;;
```

De même que pour les listes paresseux, notre imprimeur nécessite une limite qui arrête l'impression au bout d'un certain nombre de termes.

```
let rec print_série until s =
 open_hovbox 1;
 let c = s.Constante in
 if until == 0 then print_num c else
 let rest = ref s in

 let zéro = not (print_first_terme !rest) in
 if not zéro then print_space();
 for i = 1 to until do
  rest := reste_de_série !rest;
  let delim = if i == 1 & zéro then "" else "+ " in
  if print_terme delim i !rest then print_space()
 done;
 print_string "+ O(z^"; print_int (succ until);
```

```
 print_string ")";
 close_box();;
```

Nous pouvons maintenant définir (récursivement) les séries entières des fonctions sinus et cosinus :

```
let rec sinus =
 {Constante = zéro;
  Reste = Gelé (function () -> integre_depuis un cosinus)}
and cosinus =
 {Constante = un;
  Reste = Gelé (function () -> integre_depuis un (opposée_de_série sinus))};;
sinus : série = {Constante=0; Reste=Gelé <fun>}
cosinus : série = {Constante=1; Reste=Gelé <fun>}
```

Et cette définition très naïve est cependant effective :

```
print_série 10 sinus;;
z - 1/6z^3 + 1/120z^5 - 1/5040z^7 + 1/362880z^9 + O(z^11)- : unit = ()
print_série 10 cosinus;;
1 - 1/2z^2 + 1/24z^4 - 1/720z^6 + 1/40320z^8 - 1/3628800z^10 + O(z^11)
```

Notez une fois de plus la magie de la paresse : les deux séries se développent mutuellement à la demande.
Un joli théorème permet de deviner facilement la valeur de la série `s` suivante :

```
let s = add_série (mult_série sinus sinus) (mult_série cosinus cosinus);;
```

Si le dernier mot reste évidemment au mathématicien, on est heureux de constater que la machine calcule une approximation sans faille du résultat théorique :

```
print_série 10 s;;
1 + O(z^11)- : unit = ()
```

Un peu de combinatoire des mots

Mots de Lukasiewicz, nombres de Catalan

Définition et propriétés des mots de Lukasiewicz

Les mots de Lukasiewicz sont des mots sur l'alphabet $\{-1, 1\}$, qu'on notera $u = (u_0, u_1, \ldots, u_{n-1})$ ($n = \ell(u)$ est la longueur du mot), et qui vérifient les deux propriétés suivantes :

$$\sum_{k=0}^{p} u_k = \begin{cases} \geq 0, & \text{si } 0 \leq p \leq n-2; \\ -1, & \text{si } p = n-1. \end{cases}$$

On note ici $\mathcal{M}$ (*resp.* $\mathcal{L}$) l'ensemble des mots sur $\{-1, 1\}$ (*resp.* l'ensemble des mots de Lukasiewicz).

Notons que -1 est l'unique mot de Lukasiewicz de longueur 1, qu'il n'y en a pas de longueur 2, et que $(1,-1,-1)$ est l'unique mot de Lukasiewicz de longueur 3.
On appellera *capsule* d'un mot $u \in \mathcal{M}$ un sous-mot de la forme $(+1)(-1)(-1)$.
On définit un *décapsuleur* :

$$\rho : \left\{ \begin{array}{lcl} \mathcal{M} & \longrightarrow & \mathcal{M} \\ u & \longmapsto & \begin{cases} u, & \text{si } u \text{ ne contient pas de capsule ;} \\ (u_0, \ldots, u_{i-1}, u_{i+2}, \ldots, u_{n-1}), & \text{si } (u_i, u_{i+1}, u_{i+2}) = (+1,-1,-1) \\ & \text{est la première capsule de } u. \end{cases} \end{array} \right.$$

Comme $\ell(\rho(u)) \leq \ell(u)$, on peut définir $\rho^\star$ qui associe à un mot u la limite de la suite $(\rho^n(u))$.
On démontre alors de façon élémentaire que l'on a l'équivalence :

$$\forall u \in \mathcal{M}, u \in \mathcal{L} \Longleftrightarrow \rho^*(u) = (-1).$$

En outre on peut prouver que si u et v sont des mots de Lukasiewicz, alors $(1) \sharp u \sharp v$ est un mot de Lukasiewicz ($\sharp$ dénote ici l'opérateur de concaténation des mots). Mieux, on a la décomposition inverse : tout mot de Lukasiewicz de longueur au moins égale à 3 est de la forme $(1) \sharp u \sharp v$ où u et v sont de Lukasiewicz.
C'est ce qui fait penser à imaginer une bijection Φ de $\mathcal{L}$ sur l'ensemble $\mathcal{B}$ des arbres binaires (définis en Caml par `type arbre = Vide | Nœud(arbre,arbre)`), et qu'on définit ainsi :

$$\Phi(\texttt{Vide}) = (-1), \qquad \text{et} \qquad \Phi(\texttt{Nœud}(g,d)) = (1) \sharp \Phi(g) \sharp \Phi(d).$$

On vérifie que Φ est bien bijective.
On trouvera dans les programmes 1 et 2 suivants, pages 10 et 11, les fonctions Caml correspondant à cette étude.

Nombres de Catalan

On note ici a_n le nombre de mots de Lukasiewicz de longueur n. On a donc $a_0 = 0$, $a_1 = 1$, $a_2 = 0$, $a_3 = 1$, et on traduit la décomposition décrite ci-dessus des mots de Lukasiewicz par la relation de récurrence :

$$\forall n \geq 3, a_n = \sum_{i+j=n-1} a_i a_j.$$

Soit $S(z) = \sum_{n=0}^{+\infty} a_n z^n$ la série génératrice correspondante. Les relations précédentes se traduisent par :

$$S(z)^2 = \sum_{n=0}^{+\infty} (\sum_{i+j=n} a_i a_j) z^n = \sum_{n=2}^{+\infty} a_{n+1} z^n,$$

donc $S(z) = z + zS(z)^2$.

Programme 1 Les mots de Lukasiewicz

```
exception Not_Lukasiewicz ;;

let rec somme m k =
    if k = 0 then 0
    else (hd m) + somme (tl m) (k - 1) ;;

let rec rho = function
    | 1 :: -1 :: -1 :: reste -> true, -1 :: reste
    | x :: reste -> let flag, m' = rho reste in flag, x::m'
    | m -> false, m ;;

let rec rho_étoile m =
    match rho m with
        | false, m' -> m'
        | true,  m' -> rho_étoile m' ;;

let est_Lukasiewicz m =
    match rho_étoile m with
        | [-1] -> true
        | _ -> false ;;

let rec décompose suffixe =
    let rec décomp_rec m s =
        match m with
            | n :: m' ->    if n + s = -1 then [n] , m'
                            else    let g,d = décomp_rec m' (n + s)
                                    in
                                    n :: g , d
            | _ -> raise Not_Lukasiewicz
    in
    décomp_rec suffixe 0 ;;

type arbre = Feuille | Nœud of arbre * arbre ;;

let rec Lukasiewicz_of_arbre = function
    | Feuille -> [-1]
    | Nœud(g,d)
         -> 1 :: (Lukasiewicz_of_arbre g) @ (Lukasiewicz_of_arbre d) ;;

let rec arbre_of_Lukasiewicz = function
    | [-1] -> Feuille
    | 1 :: reste -> let gauche,droit = décompose reste
                    in
                    Nœud((arbre_of_Lukasiewicz gauche),
                         (arbre_of_Lukasiewicz droit))
    | _ -> raise Not_Lukasiewicz ;;
```

Programme 2 Dessin d'arbres et mots de Lukasiewicz

```
#open "graphics" ;;

open_graph "" ;;

let rec profondeur_arbre = function
    | Feuille -> -1
    | Nœud(g,d) -> 1 + max (profondeur_arbre g) (profondeur_arbre d) ;;

let profondeur_Lukasiewicz m = profondeur_arbre (arbre_of_Lukasiewicz m) ;;

let rotation (x,y) = (y-x+240,276-x-y) ;;

let point x y = let x',y' = rotation(x,y) in fill_circle x' y' 2 ;;
let va_à x y = let x',y' = rotation(x,y) in moveto x' y' ;;
let tracer x y = let x',y' = rotation(x,y) in lineto x' y' ;;

let rec deux_puissance = function
    | 0 -> 1
    | n -> let x = deux_puissance (n/2)
           in
           if n mod 2 = 0 then x * x else 2 * x * x ;;

let dessine mot =
    let rec dessin_rec a x0 y0 =
        point x0 y0 ;
        match a with
            | Feuille -> ()
            | Nœud(g,d)
               -> let delta =
                          3 * (deux_puissance (profondeur_arbre a))
                  in
                  begin
                      dessin_rec g (x0 + delta) y0 ;
                      dessin_rec d x0 (y0 + delta) ;
                      va_à (x0 + delta) y0 ;
                      tracer x0 y0 ;
                      tracer x0 (y0 + delta)
                  end
    in
    dessin_rec (arbre_of_Lukasiewicz mot) 0 0 ;;
```

On résout (comme on le fait d'habitude avec les séries génératrices), et on obtient

$$S(z) = \frac{1 \pm \sqrt{1-4z^2}}{2z} = \frac{1-\sqrt{1-4z^2}}{2z}.$$

Or on a classiquement :

$$\sqrt{1-4z^2} = 1 - \sum_{n=1}^{+\infty} \frac{1}{2n-1}\binom{2n}{n} z^{2n},$$

et on en déduit le développement de $S(z)$:

$$S(z) = \sum_{n=1}^{+\infty} \frac{\binom{2n}{n}}{2(2n-1)} z^{2n-1}, \quad \text{et} \quad a_{2n} = 0,\ a_{2n+1} = \frac{\binom{2n+2}{n+1}}{2(2n+1)} = \frac{1}{n+1}\binom{2n}{n}.$$

Ces nombres ont été baptisés nombres de Catalan.

Mots de Lyndon, ordre lexicographique

On considère cette fois les mots sur l'alphabet $\{0, 1\}$. Leur ensemble sera noté ici $\mathcal{M}$. On définit classiquement l'ordre lexicographique sur l'ensemble $\mathcal{M}$. Il s'agit d'un ordre total sur $\mathcal{M}$, que nous noterons ici $\preceq$.
Si $u = (u_0, u_1, \ldots, u_{n-1}) \in \mathcal{M}$, on notera $u^{\circ(k)}$ le mot obtenu à partir de u par rotation : $u^{\circ(k)} = u^{\circ(k')}$ dès que $k \equiv k'$ $[n]$ et, si $0 \leq k \leq n-1$, $u^{\circ(k)} = (u_k, u_{k+1}, \ldots, u_{n-1}, u_0, u_1, \ldots, u_{k-1})$.
Un mot u est dit mot de Lyndon si pour tout entier k on a $u \preceq u^{\circ(k)}$.
On démontre alors que tout mot de Lyndon u de longueur au moins égale à 2 s'écrit sous la forme $u = u_1 \sharp u_2$ où u_1 et u_2 sont deux mots de Lyndon tels que $u_1 \preceq u_2$. Réciproquement, si u_1 et u_2 sont deux mots de Lyndon tels que $u_1 \preceq u_2$, alors $u_1 \sharp u_2$ est effectivement un mot de Lyndon.
On peut aussi définir ce qu'on appelle une *factorisation de Lyndon* d'un mot u quelconque : il s'agit d'une décomposition $u = u^0 \sharp u^1 \sharp \cdots \sharp u^k$, où chacun des u^i est un mot de Lyndon et où de plus $u^k \preceq u^{k-1} \preceq \cdots \preceq u^1 \preceq u^0$.
Un algorithme de décomposition est le suivant :

1. on part de la décomposition $u = (u_0) \sharp (u_1) \sharp \cdots \sharp (u_{n-1})$ en mots de longueur 1 ;

2. on cherche dans la décomposition courante $u = u^0 \sharp u^1 \sharp \cdots \sharp u^k$ s'il existe un indice j tel que $u^j \preceq u^{j+1}$. Si oui, on passe à 3, sinon on a terminé ;

3. on remplace la décomposition $u = u^0 \sharp u^1 \sharp \cdots \sharp u^k$ par la décomposition $u = v^0 \sharp v^1 \sharp \cdots \sharp v^{k-1}$ où $v^i = u^i$ si $i < j$, $v^j = u^j \sharp u^{j+1}$, et $v^i = u^{i+1}$ si $i > j$, et on retourne à 2.

Par exemple, la décomposition de Lyndon du mot (0101001100100) est (01)(01)(0011)(001)(0)(0).

Programme 3 Les mots de Lyndon

```
exception Not_Lyndon ;;

type comparaison = Inférieur | Égal | Supérieur ;;

let rec ordre u v = match u,v with
    | [],[] -> Égal
    | [],_ -> Inférieur
    | _,[] -> Supérieur
    | u0 :: u' , v0 :: v'
        -> if u0 = v0 then ordre u' v'
           else if u0 < v0 then Inférieur else Supérieur ;;

let rec est_préfixe a mot = match a with
    | [] -> true
    | a0 :: a' -> match mot with
                    | m0 :: m' when a0 = m0 -> est_préfixe a' m'
                    | _ -> false ;;

let est_suffixe a mot =
    let rec shift l = function
        | 0 -> l
        | n -> shift (tl l) (n - 1)
    in
    let la,lm = (list_length a),(list_length mot)
    in
    if la > lm then false
    else a = (shift mot (lm - la)) ;;

let rotation mot = (tl mot) @ [ hd mot ] ;;

let est_Lyndon_def mot =
    let n = list_length mot
    and m = ref mot
    in
    try
        for i = 1 to n  do
            m := rotation ! m ;
            if (ordre mot ! m) = Supérieur then raise Not_Lyndon
        done ;
        true
    with Not_Lyndon -> false ;;

let est_Lyndon mot =
    let rec test = function
        | [] -> true
        | (a :: q) as m' -> (Inférieur = ordre mot m') && (test q)
    in
    test (tl mot) ;;
```

Programme 4 Factorisation des mots de Lyndon

```
let factorisation_de_Lyndon mot =
    let rec réduit = function
        | (a :: b :: q) as ll
            ->  ( if (ordre a b) = Inférieur then
                        let m,_ = réduit ((a @ b) :: q) in m,true
                  else
                        match réduit (b :: q) with
                            | m,true -> réduit (a :: m)
                            | m,false -> ll,false                           )
        | ll -> ll,false
    in
    match réduit (map (function x -> [x]) mot) with ll,_ -> ll ;;

let Lyndon =
    let rec insertion x = function
        | [] -> [x]
        | y :: q -> match ordre x y with
                        | Inférieur -> x :: y :: q
                        | Égal -> y :: q
                        | Supérieur -> y :: (insertion x q)
    in
    let rec map_accu f accu = function
        | [] -> accu
        | x :: q -> match f(x) with
                        | [] -> map_accu f accu q
                        | y -> map_accu f (insertion y accu) q
    in
    let compose_un x1 l2 accu =
        map_accu (function x2 -> if Inférieur = ordre x1 x2
                                  then x1 @ x2 else [])
                 accu l2
    in
    let rec compose l1 l2 accu =
        match l1 with
            | [] -> accu
            | x1 :: q -> compose q l2 (compose_un x1 l2 accu)
    in
    let rec compose_tout f n k accu =
        if k < n then compose_tout f n (k+1) (compose (f k) (f (n-k)) accu)
        else accu
    in
    let mémoire = ref [ (0,[]) ; (1,[ [0] ; [1] ]) ]
    in
    let rec f n =
        try assoc n !mémoire
        with Not_found -> let res = compose_tout f n 1 []
                          in
                          mémoire := (n,res) :: !mémoire ;
                          res
    in
    f ;;
```

Permutations

La représentation choisie ici

Une idée naturelle pour définir une permutation suit la définition mathématique : une permutation est une application σ de $\{1, \ldots, n\}$ dans lui-même. Un type Caml adapté est donc

```
type permutation = int * (int -> int) ;;
```

dans la mesure où on définira une permutation comme le couple (n, σ).
En écrivant sur cette base la fonction qui à σ associe σ^{-1}, sa réciproque, je me suis aperçu que j'étais conduit à passer par l'intermédiaire de la représentation vectorielle des permutations : peut-être y a-t-il une solution élégante, qui reste dans l'esprit de la représentation fonctionnelle des permutations, mais je ne l'ai pas trouvée.
On choisit donc ici de représenter une permutation de $\{1, \ldots, n\}$ par un vecteur d'entiers : la permutation σ est représentée par le vecteur d'entiers

$$v(\sigma) = [|\ \sigma(1); \sigma(2); \ldots; \sigma(n)\ |].$$

On n'aura pas oublié que Caml indexe les vecteurs à partir de 0. Ainsi donc on récupérera $\sigma(i)$ en évaluant $v.(i-1)$. Cela justifie les deux fonctions fondamentales suivantes :

```
let image v k = v.(k - 1)
and affecte v k x = v.(k-1) <- x ;;
```

De cette façon, on écrira par exemple ainsi la permutation identique :

```
let identité n =
    let v = make_vect n 1
    in
    for i = 1 to n do affecte v i i done ;
    v ;;
```

Quelques fonctions simples

Voici tout d'abord de quoi vérifier qu'un vecteur d'entiers donné représente bien une permutation :

```
let rec intervalle_d'entiers i j =
    if i > j then []
    else i :: (intervalle_d'entiers (i+1) j) ;;

let est_permutation v =
    let n = vect_length v
    in
    let un_a_n = intervalle_d'entiers 1 n
    in
    let rec est_ok = function
        | [] -> true
        | a :: q -> (not (mem a q)) && (mem a un_a_n) && (est_ok q)
    in
    est_ok (list_of_vect v) ;;
```

On pourra reprocher à cette version d'être quadratique. Une version linéaire est par exemple la suivante :

```
let est_permutation v =
    let n = vect_length v
    in
    let test = make_vect n false
    in
    try for i = 0 to (n-1)
        do let j = v.(i) - 1
           in
           if test.(j) then failwith "doublon"
                       else test.(j) <- true
        done ;
        true
   with _ -> false ;;
```

On observera que le filtrage d'exception `try ... with _ -> false` permet non seulement de rattraper l'erreur `Failure "doublon"` mais aussi les erreurs que déclencherait l'appel à `test.(j)` dans le cas où le vecteur de taille n contiendrait un élément supérieur à n.
On écrit ensuite naturellement la composition des permutations :

```
let compose v v' =
    let n,n' = (vect_length v),(vect_length v')
    in
    if n = n' then
        let w = make_vect n 1
        in
        for i = 1 to n do affecte w i (image v (image v' i)) done ;
        w
    else
        failwith "Composition de deux permutations de tailles différentes" ;;
```

L'orbite d'un entier k pour la permutation σ est l'ensemble $\{\sigma^j(k), j \in \mathbb{N}\}$. On l'obtient facilement ainsi :

```
let orbite v k =
    let rec augmente_orbite k liste =
        if mem (image v k) liste then liste
        else augmente_orbite (image v k) ((image v k) :: liste)
    in
    augmente_orbite k [ k ] ;;
```

Pour rechercher les points fixes d'une permutations, il paraît naturel d'écrire la fonction que j'ai baptisée `select` (et qui me semble manquer à la bibliothèque standard de Caml) qu'on peut définir ainsi :

```
let rec select prédicat = function
    | [] -> []
    | a :: q -> if (prédicat a) then a :: (select prédicat q)
                                else (select prédicat q) ;;
```

et qui choisit ceux des éléments d'une liste qui vérifient un prédicat (*id est* une fonction de type `'a -> bool`) donné.

On a alors tout simplement :

```
let points_fixes v =
    let n = vect_length v
    in
    select (function x -> x = (image v x)) (intervalle_entier 1 n) ;;
```

Notre choix d'une représentation vectorielle des permutations rend élémentaire l'écriture de la réciproque :

```
let réciproque v =
    let n = vect_length v
    in
    let w = make_vect n 1
    in
    for i = 1 to n do affecte w (image v i) i done ;
    w ;;
```

Pour terminer dans cette petite collection, je vous propose maintenant de décomposer une permutation en cycles :

```
let rec ôte x = function
    | [] -> []
    | a :: q when a = x -> q
    | a :: q -> a :: (ôte x q) ;;
```

fait comme la fonction `subtract` de la bibliothèque standard de Caml, mais ne supprime d'une liste que la première occurrence d'un objet `x` donné, ce qui suffit dans le cas des permutations.
On est en mesure de construire la décomposition en cycles (on renvoie la liste des cycles, eux-mêmes représentés par des listes) :

```
let cycles v =
    let rec un_cycle x0 x candidats =
        if (image v x) = x0 then [ x0 ] , (ôte x candidats)
        else
            let queue,c' = un_cycle x0 (image v x) (ôte x candidats)
            in
            x :: queue,c'
    in
    let rec épuise = function
        | [] -> []
        | a :: _ as liste ->    let cycle,candidats = un_cycle a a liste
                                in
                                cycle :: (épuise candidats)
    in
    épuise (list_of_vect v) ;;
```

Le problème de la génération des permutations

Génération récursive de l'ensemble des permutations

```
let échange v i j =
    let x = v.(i)
    in
    v.(i) <- v.(j) ;
    v.(j) <- x ;;
```

permet bien entendu d'échanger deux éléments d'un vecteur.
On écrit maintenant l'analogue d'un `do_list f` S_n : étant donnés une fonction `f : int -> unit` et un entier `n`, `do_list f n` applique la fonction à toutes les permutations de $\{1, \ldots, \mathtt{n}\}$ tour à tour. La fonction construit récursivement — par échanges successifs — les permutations à partir de la permutation identique. C'est `perm_rec` qui fait tout le travail, en échangeant tour à tour l'élément i avec tous les suivants. Quand elle est appelée avec $i = \mathtt{n} - 1$ c'est qu'on a une permutation toute prête, qu'on passe alors à la fonction f.

```
let applique_aux_permutations f n =
    let v = vect_of_list (intervalle_d'entiers 1 n)
    in
    let rec perm_rec i =
        if i = n - 1 then f v
        else
            for j = i to n - 1 do
                échange v i j ;
                perm_rec (i+1) ;
                échange v i j
            done
    in
    perm_rec 0 ;;
```

On vérifie que chaque permutation est engendrée une fois et une seule : cette fonction a une complexité $O(n!)$.
Application : on souhaite imprimer la liste complète de toutes les permutations de $\{1, 2, \ldots, n\}$ pour un n fixé ; on commencera par définir une fonction d'impression d'une permutation, comme celle-ci (qui est évidemment très rudimentaire) :

```
let print_permutation v =
    for i = 0 to (vect_length v) - 1 do
        print_int v.(i) ;
        print_char ‘ ‘
    done ;
    print_newline () ;;
```

L'invocation suivante permet alors d'imprimer tour à tour toutes les permutations :

```
let toutes_les_permutations = applique_aux_permutations print_permutation ;;
```

Une méthode tout à fait différente

Remarquant que l'on peut définir un ordre sur S_n en rangeant les permutations dans l'ordre lexicographique, on souhaite écrire une fonction qui calcule

la permutation qui succède à une permutation donnée, ou qui déclenche une exception dans le cas où la permutation fournie est la dernière (c'est-à-dire en l'occurrence la permutation $(n, n-1, \ldots, 3, 2, 1)$).
Expliquons sur un exemple l'algorithme utilisé. Soit $\sigma = (346521)$ une permutation élément de S_6, la suivante est $\sigma' = (351246)$. Comment la calculer.
On recherche tout d'abord le plus grand suffixe décroissant de σ : c'est ici (6521). La fonction `début_suffixe` ci-dessous renvoie donc 1, l'indice de 4, qui précède le suffixe. C'est elle qui sait déclencher l'exception `No_more_permutation` qui signale qu'on a fourni la dernière permutation de S_n, qu'elle reconnaît facilement.
On retourne alors le suffixe, obtenant dans notre exemple la permutation (341256). Il suffit pour terminer d'échanger le 4 qui balisait le début du préfixe avec le premier (de gauche à droite) élément du suffixe qui lui est supérieur (ici il s'agit de 5) : on obtient bien $\sigma' = (351246)$.

```
exception No_more_permutation ;;

let permutation_suivante v =
    let n = vect_length v
    in
    let rec début_suffixe i =
        if i <= 0 then raise No_more_permutation
        else
        if v.(i) < v.(i-1) then début_suffixe (i-1)
        else i-1
    and retourne_suffixe a b =
        if a < b then
        begin
            échange v a b ;
            retourne_suffixe (a + 1) (b - 1)
        end
    and place k j =
        if v.(j) > v.(k) then échange v j k
        else place k (j + 1)
    and modifie_suffixe k =
            retourne_suffixe (k + 1) (n - 1) ;
            place k (k + 1)
    in
    modifie_suffixe ( début_suffixe (n - 1) ) ;;
```