La lettre de Caml numéro 1

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

octobre 1995

Édito

Et voilà : j'ai craqué. En attendant le service espéré du Ministère, je me suis abonné à un service commercial pour avoir un accès à Internet. Vous y gagnez (et moi un peu moins, m'enfin presque) : vous pouvez désormais télécharger cette lettre de Caml sur le serveur de l'INRIA, qui m'héberge grâce à la générosité de cette institution et surtout de Pierre Weis, que je remercie ici. L'adresse de la lettre est la suivante :

http: //pauillac.inria.fr/caml/lettre_de_caml/index.html

On peut télécharger séparément les programmes Caml, si l'on préfère. Je pense pouvoir aussi mettre à votre disposition la démo. de BBEdit, et les petites extensions dont je parlais le mois dernier. Bien entendu, Pierre Weis a d'autres charges de travail, et les délais d'installation sur le serveur de l'INRIA en dépendent directement.
D'autre part Alain Chillès, pour l'UPS, m'a proposé de profiter du réseau de cette association pour la diffusion de la Lettre de Caml. Qu'il en soit lui aussi remercié ici.
Quelques uns d'entre vous ont répondu à mon appel et m'ont fait part de leurs impressions sur le numéro 0. Je les remercie, et espère toujours vos contributions à tous.
N'hésitez pas à me demander d'aborder tel ou tel sujet qui vous intéresserait davantage que ceux que j'aborde.
Merci de votre indulgence.

Table des matières

Un convertisseur graphique

Pour les utilisateurs de PC

Une fois n'est pas coutume, nous nous intéresserons en particulier aux utilisateurs de PC.
La bibliothèque graphique de Caml définit un type `image` pour la représentation interne de zones rectangulaires de la fenêtre graphique. Une autre représentation de la même image correspond au type `color vect vect`, qui est bien adaptée à des traitements de l'image, et nous concernera donc directement en TIPE. La bibliothèque fournit les deux fonctions de conversion :

`make_image : color vect vect -> image`
et
`dump_image : image -> color vect vect`

On dessine une image par `draw_image : image -> int -> int -> unit` qui attend les coordonnées du coin inférieur gauche (et non droit — en tout cas sur mon Mac — comme indiqué dans le manuel de référence) de l'image à dessiner dans l'écran graphique.
Denis Monasse *(enseignant au lycée Louis-le-Grand, email : `monasse@cicrp.jussieu.fr`)* nous propose la fonction de conversion (voir pages suivantes), qui prend un fichier `.bmp` créé sous Windows et crée un fichier `.clg`, soit *Caml Light Graphics*. Il nous propose aussi une fonction qui charge un fichier `.clg` et fournit un objet de type `color vect vect`, à partir duquel on pourra procéder à tous les traitements voulus.
Un exemple typique d'utilisation est le suivant :

```
#open "graphics" ;;

open_graph "" ;;

bmp2clg "mon_fichier" ;;
let mon_image = make_image (charge_clg "mon_fichier") ;;

draw_image mon_image 0 0 ;;
```

Et pour les fans de Mac...

Je vous propose la manipulation suivante pour récupérer une image.

1. Lancez Photoshop.

2. Ouvrez un fichier en 72 points par pouce, et en mode RGB; peu importe son format. Attention ! la fenêtre graphique Caml est — hélas — limitée à une taille de 480×280.

3. Dans le menu **Mode**, choisissez le mode couleurs indexées et choisissez 256 couleurs (ou 8 bits/pixel, c'est pareil). Je vous recommande le mode **Diffusion**.

4. Enregistrez votre fichier (avec une extension `.bmp`, c'est mieux) dans le format BMP, Windows, 8 bits/pixel, sans compression.

5. Voilà, c'est fini.

Programme 1 Conversion `bmp` (*Windows*) vers `clg` (pour Caml)

```
(*********************************************************************
Syntaxe : bmp2clg "nom_fichier"
     ou : bmp2clg "nom_fichier.bmp"
crée un fichier nommé "nom_fichier.clg"
*********************************************************************)

let nom_sans_extension nom ext =
    let n = string_length nom
    and n' = string_length ext
    in
    if n > n' && eq_string ("." ^ ext) (sub_string nom (n-n'-1) (n'+1))
        then sub_string nom 0 (n-n'-1)
    else nom ;;

let bmp2clg fichier_bmp =
    let nom = nom_sans_extension fichier_bmp "bmp"
    in
    let canal_in = open_in  (nom ^ ".bmp")
    and canal_out = open_out (nom ^ ".clg")
    in
    let lit_deux_octets décalage =
    begin
        seek_in canal_in décalage ;
        let poids_faible = input_byte canal_in
        in
        let poids_fort = input_byte canal_in
        in
        poids_faible + 256 * poids_fort
    end
    in
    let largeur = 4 * ((lit_deux_octets(18) + 3)/4)
    in
    output_binary_int canal_out largeur ;
    let hauteur = lit_deux_octets(22)
    in
    output_binary_int canal_out hauteur ;
    print_string "largeur = " ; print_int largeur ;
    print_string " , hauteur = " ; print_int hauteur ;
    print_newline () ;
    let palette=make_vect 256 0
    in
    let lit_couleur () =
        let b = input_byte canal_in
        in
        let g = input_byte canal_in
        in
        let r = input_byte canal_in
        in
        let _ = input_byte canal_in
        in
        rgb r g b
    in
    seek_in canal_in 54 ;
    for i = 0 to 255 do palette.(i) <- lit_couleur() done ;
    for i = 1 to hauteur do
        seek_in canal_in (1078 + largeur * (hauteur-i)) ;
        for j=1 to largeur do
            output_binary_int canal_out palette.(input_byte canal_in)
        done
    done ;
    close_out canal_out ;
    close_in canal_in ;;
```

Programme 2 Chargement des fichiers `clg`

```
(*******************************************************************
Syntaxe: charge_clg "nom_fichier"
     ou: charge_clg "nom_fichier.clg"
*******************************************************************)

let charge_clg fichier_clg =
    let nom = (nom_sans_extension fichier_clg "clg") ^ ".clg" in
    let canal_in = open_in nom in
    let largeur = input_binary_int canal_in in
    let hauteur = input_binary_int canal_in in
    let image = make_matrix hauteur largeur 0
    in
    for i=0 to hauteur-1 do
        for j=0 to largeur -1 do
            image.(i).(j) <- input_binary_int canal_in
        done
    done ;
    image ;;
```

Une recherche linéaire du i-ème plus petit élément

La première fois qu'on m'en a parlé, je n'y ai pas cru !
C'est vrai, quoi ! on se donne une liste de n éléments d'un ensemble ordonné, et il s'agit de chercher le i-ème élément, *dans l'ordre croissant*. J'aurais parié sur un $O(n \ln n)$. Eh bien non : il existe une solution linéaire.
Le sujet a été rapidement abordé — et d'une façon hélas assez floue — lors du dernier stage de Luminy. Je vais tenter ici de faire le point, en Caml, bien sûr.

Description informelle de l'algorithme

On se donne une liste non triée, dont les éléments, réordonnés, s'écriraient

$$a_0 \leqslant a_1 \leqslant \cdots \leqslant a_{n-2} \leqslant a_{n-1}.$$

On cherche à trouver l'élément a_i, c'est-à-dire (en commençant la numérotation par 0 comme toujours en Caml) le i-ème dans l'ordre croissant.
On va utiliser l'habituelle *stratégie diviser pour régner*.
Supposons la liste partagée en deux morceaux autour d'un pivot : les éléments plus petits que le pivot d'une part, les éléments supérieurs ou égaux d'autre part. Si n_1 et n_2 sont les tailles des deux sous-ensembles ainsi créés, ou bien $i < n_1$ et on appliquera récursivement notre procédure à la première liste, ou bien $i > n_1$, et on appliquera récursivement notre procédure à la seconde liste, avec $(i - n_1)$.
Bien sûr, l'algorithme tournera d'autant plus vite que le pivot aura été choisi vers le milieu de la liste — mais il ne faudrait pas l'appeler récursivement avec $i = n/2$ pour trouver ce pivot !

Une bonne idée consiste à découper la liste de départ en paquets de 5 éléments, et de prendre pour pivot l'élément médian (trouvé par un appel récursif de notre procédure, cette fois) des milieux de ces paquets de 5 qu'on aura pu trier en temps constant l'un après l'autre (un tri de 5 éléments prend un temps constant, bien sûr).

Implémentation en Caml

Nous prendrons ici comme domaine de travail l'ensemble des entiers ; la transposition à un autre type est laissée au lecteur.

`découpe liste pivot` renvoie un quadruplet `(linf,n1,lsup,n2)` composé de deux listes `linf`, de taille `ninf`, et `lsup` de taille `nsup`, telles que `linf` contienne ceux des éléments de la `liste` de départ qui sont strictement plus petits que le `pivot`, et `lsup` tous les autres.

Programme 3 Division de la liste

```
let découpe liste pivot =
    let rec découpe_rec l1 n1 l2 n2 = function
              [] -> l1,n1,l2,n2
            | a :: q -> if a < pivot
                    then découpe_rec (a::l1) (n1+1) l2 n2 q
                    else découpe_rec l1 n1 (a::l2) (n2+1) q
    in découpe_rec [] 0 [] 0 liste ;;

(* par exemple :                   *)
(*      découpe [1;3;2;5;4] 3 ;;    *)
(*      s'évalue en                *)
(*      [2; 1], 2, [4; 5; 3], 3    *)
```

La fonction `quintuplifie` (voir page suivante) découpe une liste en paquets de 5 éléments. La fonction `milieux` prend une liste de paquets de 5 (ou moins, par extraordinaire) et renvoie une liste des éléments médians des paquets (sauf l'exception des petits paquets pour lesquels on se contente de renvoyer le premier élément du paquet, ce qui ne change en rien le résultat de l'algorithme).

Enfin, la fonction `nth` est analogue à celle qu'on souhaite écrire mais prend en argument une liste déjà triée, ce qui simplifie bien sûr le travail !

Programme 4 5−listes

```
let rec quintuplifie = function
      [] -> []
    | a::b::c::d::e::q -> [a;b;c;d;e] :: (quintuplifie q)
    | l -> [l] ;;

let rec milieux = function
      [] -> []
    | [a;b;c;d;e] :: q -> c :: (milieux q)
    | (a::_)::q -> a :: (milieux q) ;;

let rec nth n l = if n=0 then hd l else nth (n-1) (tl l) ;;
```

Il ne reste plus qu'à suivre l'algorithme que nous avons décrit plus haut.

On évacue le cas des petites listes (ce qui permet d'ailleurs de stopper la récursion) en ligne 36 où on appelle simplement la fonction `nth` qui attend une liste triée.
Sinon, on procède au découpage (ligne 38) par rapport au pivot, qui a été trouvé grâce à la procédure définie en lignes 27—30. Il n'y a plus qu'à faire l'appel récursif sur la partie intéressante (lignes 40—42).
Il me semble qu'on a là un programme assez clair pour un algorithme dont la traduction en Pascal aurait pu poser quelques problèmes... Mais vous êtes déjà sans aucun doute tous convaincus de tout l'intérêt du langage Caml !

Programme 5 La fonction `nth`

```
24 let rec nth_linéaire n l =
25         (* renvoie le n-ième élément de la liste l, *)
26         (* dans l'ordre, nth 0 l est donc le min    *)
27     let choisit_pivot l n =
28         let médians = milieux (map tri_insertion (quintuplifie l))
29         in
30         nth_linéaire (((n+4)/5)/2) médians
31     in
32     match l with
33       [] -> failwith "Liste vide"
34     | l ->  let nl = (list_length l)
35             in
36             if nl <= 5 then nth n (tri_insertion l)
37             else
38             let l1,n1,l2,n2 = découpe l (choisit_pivot l nl)
39             in
40             if n < n1
41             then nth_linéaire n l1
42             else nth_linéaire (n-n1) l2 ;;
43
44 (* par exemple :                            *)
45 (*      nth_linéaire 6 [1;4;2;7;3;0;5;6]    *)
46 (*      s'évalue en 6                       *)
```

Un algorithme effectivement linéaire

Le petit schéma suivant permet de rapidement conclure pour l'évaluation de l'algorithme. On a

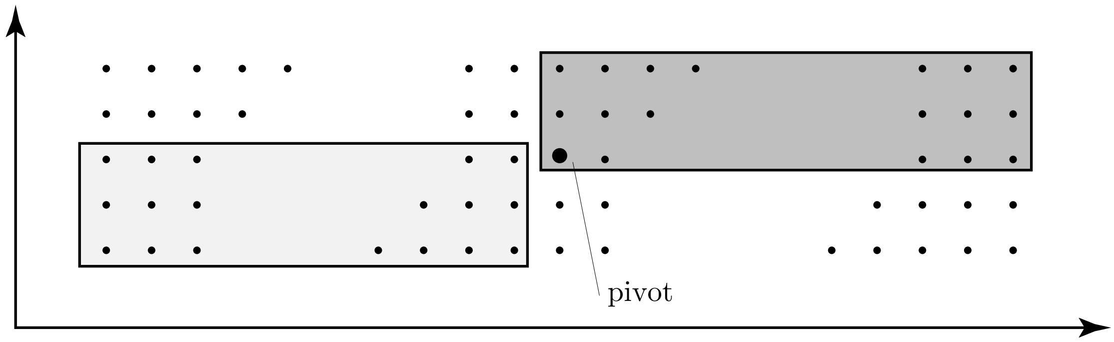

Figure 1: Pour l'évalutation de `nth_linéaire`

représenté verticalement chaque paquet de 5, rangé (à coût constant) verticalement de bas en haut : les éléments médians sont donc sur la ligne médiane. La découpe a été faite seulement sur ces éléments médians : sur la même ligne que le pivot, mais à sa gauche, sont les médians plus petits, et à sa droite les médians plus grands.
Il est alors clair que le rectangle légèrement grisé ne contient que des éléments inférieurs au pivot, et qu'il y en a $3 \times (n/10)$. Le rectangle plus foncé ne contient que des éléments supérieurs au pivot, et il y en a $3 \times (n/10)$. À cause des arrondis, Mehlhorn[1] convient de dire qu'il y a au moins $3n/11$ éléments dans chacun de ses rectangles. Or $n - 3n/11 = 8n/11$, et chacune des deux listes produites par `découpe` (les minorants et les majorants du pivot) contiennent donc au plus $8n/11$ éléments.
La récurrence vérifiée par le coût $T(n)$ de l'algorithme s'écrit donc

$$T(n) \leqslant T(\lceil n/5 \rceil) + T(8n/11) + k.n,$$

puisque qu'on n'appelle récursivement notre procédure que d'une part sur une des deux listes découpées et d'autre part sur la liste des médians, qui est de taille $\lceil n/5 \rceil$, les autres opérations étant de coût linéaire.
Je vous laisse vérifier que l'algorithme est bien linéaire.

---

[1]voir *Kurt Mehlhorn, Data structures and algorithms (1) : sorting and searching*, Springer-Verlag, 1984, pages 94 sqq

Retour sur les parseurs

Après l'exemple d'un parseur pour les expressions régulières du numéro 0, je vous propose tout simplement un analyseur d'expressions arithmétiques. On verra en particulier que la question bien gênante — quand on essaie d'écrire l'analyseur en Pascal — de la distinction du moins unaire (opposé) et du moins binaire (soustraction) disparaît complètement.
Je retiens ici les quatre opérations et l'élévation à la puissance.

Rapidement, le lexeur

Programme 6 Un lexeur pour les expressions arithmétiques

```
 1 type lexème = Entier of int | Plus | Moins | Multiplie | Divise
 2                | Puissance | ParenthèseGauche | ParenthèseDroite ;;
 3
 4 let int_of_digit c = (int_of_char c) - (int_of_char `0`) ;;
 5
 6 let rec MangeEntier flot accu = match flot with
 7     | [< '(`0`..`9` as c) >] -> MangeEntier flot (10*accu+(int_of_digit c))
 8     | [< >] -> Entier(accu) ;;
 9
10 let rec lexeur flot = match flot with
11     | [< '(` ` | `\r` | `\t` | `\n`) >] -> lexeur flot
12     | [< '`+` >] -> [< 'Plus ; (lexeur flot) >]
13     | [< '`-` >] -> [< 'Moins ; (lexeur flot) >]
14     | [< '`*` >] -> [< 'Multiplie ; (lexeur flot) >]
15     | [< '`/` >] -> [< 'Divise ; (lexeur flot) >]
16     | [< '`^` >] -> [< 'Puissance ; (lexeur flot) >]
17     | [< '`(` >] -> [< 'ParenthèseGauche ; (lexeur flot) >]
18     | [< '`)` >] -> [< 'ParenthèseDroite ; (lexeur flot) >]
19     | [< '(`0`..`9` as c) >]
20             -> [< '(MangeEntier flot (int_of_digit c)) ; (lexeur flot) >]
21     | [< >] -> [< >] ;;
```

La seule petite difficulté est dans l'écriture de la procédure `MangeEntier` qui avale tous les chiffres d'un flot en reconstituant l'entier correspondant. On aura remarqué qu'elle est appelée quand on a reconnu un entier grâce à son premier chiffre, qui est donc aussitôt avalé et supprimé du flot. On a utilisé — ce qui est très classique en programmation fonctionnelle — un argument du genre *accumulateur*, puisque l'on écrit les entiers de gauche à droite (on écrirait cent-vingt-trois sous la forme 321, on pourrait éviter tout accumulateur…).
Remarquons que notre lexeur s'arrête en fin de flux ou sur tout caractère inattendu, mais sans déclencher d'erreur. C'est, bien entendu, le rôle de la ligne 21, qu'on pourrait supprimer si l'on voulait rejeter les caractères illégaux.
Enfin, notons son type : `lexeur : char stream -> lexème stream`.

Traitement de la grammaire

La grammaire *naturelle* est la suivante :

```
Expression ::= entier
             | ( Expression )
             | - Expression
             | Expression + Expression
             | Expression - Expression
             | Expression * Expression
             | Expression / Expression
             | Expression ^ Expression
```

Notre premier travail est d'ôter toute ambigüité à cette grammaire, en utilisant, comme la dernière fois, les priorités des opérateurs.
On obtient :

```
E ::= F + E | F - E | F

F ::= G * F | G / F | G

G ::= H ^ G | H

H ::= -I | I

I ::= ( E ) | entier
```

Il ne reste plus, à cause du mode de filtrage des flots de Caml, qu'à factoriser cette grammaire à gauche, ce qui fournit notre grammaire définitive.

```
E  ::= F E'
E' ::= + E | - E | ∅
F  ::= G F'
F' ::= * F | / F | ∅
G  ::= H G'
G' ::= ^ G | ∅
H  ::= -I | I
I  ::= ( E ) | entier
```

Le parseur

On trouvera page précédente l'implémentation Caml du parseur correspondant à la grammaire ci-dessus.

On notera que, comme dans l'exemple de la lettre numéro 0, il n'y a presque rien à faire. Simplement, pour pouvoir représenter les réductions à ∅, il est nécessaire que `parseur_E'`, `parseur_F'` et `parseur_G'` renvoient un flot d'expressions alors que toutes les autres procédures renvoient directement une expression. Ou bien le flot renvoyé est vide (exemple : la ligne 42), ou bien c'est une expression dont seul le dernier argument est significatif (exemple : la ligne 40).

Programme 7 Un parseur pour les expressions arithmétiques

```
type expression = N of int
        | Opposé of expression
        | Somme of expression*expression
        | Différence of expression*expression
        | Produit of expression*expression
        | Quotient of expression*expression
        | Élévation of expression*expression ;;

exception Syntax_error ;;

let rec parseur_E flot = match flot with
    | [< parseur_F f ; parseur_E' e' >]
            -> match e' with
                | [< 'Somme(_,e) >] -> Somme(f,e)
                | [< 'Différence(_,e) >] -> Différence(f,e)
                | [< >] -> f
    | [< >] -> raise Syntax_error
and parseur_E' flot = match flot with
    | [< 'Plus ; parseur_E e >] -> [< 'Somme(N(0),e) >]
    | [< 'Moins ; parseur_E e >] -> [< 'Différence(N(0),e) >]
    | [< >] -> [< >]
and parseur_F flot = match flot with
    | [< parseur_G g ; parseur_F' f' >]
            -> match f' with
                | [< 'Produit(_,e) >] -> Produit(g,e)
                | [< 'Quotient(_,e) >] -> Quotient(g,e)
                | [< >] -> g
    | [< >] -> raise Syntax_error
and parseur_F' flot = match flot with
    | [< 'Multiplie ; parseur_F f >] -> [< 'Produit(N(0),f) >]
    | [< 'Divise ; parseur_F f >] -> [< 'Quotient(N(0),f) >]
    | [< >] -> [< >]
and parseur_G flot = match flot with
    | [< parseur_H h ; parseur_G' g' >]
            -> match g' with
                | [< 'Élévation(_,e) >] -> Élévation(h,e)
                | [< >] -> h
    | [< >] -> raise Syntax_error
and parseur_G' flot = match flot with
    | [< 'Puissance ; parseur_G g >] -> [< 'Élévation(N(0),g) >]
    | [< >] -> [< >]
and parseur_H flot = match flot with
    | [< 'Moins ; parseur_I i >] -> Opposé(i)
    | [< parseur_I i >] -> i
    | [< >] -> raise Syntax_error
and parseur_I flot = match flot with
    | [< 'ParenthèseGauche ; parseur_E e ; 'ParenthèseDroite >] -> e
    | [< 'Entier(n) >] -> N(n)
    | [< >] -> raise Syntax_error ;;

let parseur s = parseur_E (lexeur (stream_of_string s)) ;;
```

L'évaluateur

Il ne reste plus qu'à évaluer les expressions arithmétiques ainsi décortiquées, ça n'est vraiment pas difficile !

Programme 8 Évaluation des expressions arithmétiques

```
exception Power_error of string ;;

let carré n = n*n ;;
let impair n = (n mod 2) <> 0 ;;

let rec puissance a b =
    if b < 0 then raise (Power_error "exposants négatifs interdits")
    else if a = 0 then
            if b = 0 then raise (Power_error "0^0 n'existe pas")
            else 0
    else if b = 0 then 1
    else if impair b then a * carré(puissance a (b/2))
    else carré(puissance a (b/2)) ;;

let rec évaluation = function
    | N(n) -> n
    | Opposé(e) -> - (évaluation e)
    | Somme(e,f) -> (évaluation e) + (évaluation f)
    | Différence(e,f) -> (évaluation e) - (évaluation f)
    | Produit(e,f) -> (évaluation e) * (évaluation f)
    | Quotient(e,f) -> (évaluation e) / (évaluation f)
    | Élévation(e,f) -> puissance (évaluation e) (évaluation f) ;;

let évalue s = évaluation (parseur s) ;;
```

Il ne vous reste plus qu'à essayer ces programmes. Bien sûr, je peux vous les fournir à la demande, ce qui vous évitera de les retaper...

Le ~~théorème~~ problème du point fixe

Considérons le problème suivant : on se donne un graphe symétrique, et un sommet v_0 de ce graphe. On dispose pour tout sommet v du graphe de la liste `voisins`(v) des sommets qui lui sont reliés par une arête. On cherche la liste de tous les sommets reliés à v_0. Pour résoudre ce problème, il suffit d'ajouter petit à petit les voisins des voisins, etc., jusqu'à ce que la liste de sommets obtenue ne grandisse plus. C'est donc un problème de point fixe.
Dans la suite, nous représenterons les ensembles par des listes. Rappelons que la bibliothèque standard de Caml fournit les fonctions `union`, `subtract`, `intersect` et `mem`.
Soit donc `f : 'a -> 'a list`. On cherche, étant donnée une liste initiale `l0 : 'a list`, à construire la liste *limite* de la suite $\mathtt{l}_n$ définie par la récurrence $\mathtt{l}_{n+1} = \mathtt{l}_n \cup \bigcup_{a \in \mathtt{l}_n} \mathtt{f(a)}$.

Cette limite sera point fixe de l'équation `l = flat_union_map f l`, où `flat_union_map` est définie en Caml de la façon suivante :

```
let rec flat_union_map f = function
                | [] -> []
                | a :: q -> union (f a) (flat_union_map f q) ;;
```

Ce problème revient de façon récurrente, puisqu'il est habituel de chercher des clôtures transitives de certaines relations : on verra qu'il est incontournable dans le cas des automates finis, pour la suppressions des ε-transitions.
On souhaite donc écrire une fonction de recherche du point fixe, à savoir `point_fixe` de type `('a -> 'a list) -> 'a list -> 'a list` ; l'appel standard sera de la forme `point_fixe f l`.
Voilà tout d'abord une solution *basique*. On commence par définir l'inclusion et l'égalité de deux ensembles : `contient l1 l2` renverra `true` si et seulement si `l1` $\supset$ `l2`, `même_ensemble` testera l'égalité.

```
let contient l1 = contient_rec where rec
        contient_rec = function
             [] -> true
           | a :: q -> (mem a l1) && (contient_rec q) ;;

let même_ensemble l1 l2 = (contient l1 l2) && (contient l2 l1) ;;
```

On est en mesure d'écrire un calcul naïf du point fixe:

```
let point_fixe f l0 =
    let image = flat_union_map f
    in
    let rec point_fixe_rec l0 =
         let l1 = image l0
         in
         if contient l0 l1 then l0
         else point_fixe_rec (union l0 l1)
    in
    point_fixe_rec l0 ;;
```

Pas très futé, ne trouvez-vous pas ? En effet, partant de `l0`, on calcule une première fois les images correspondantes par `f`, mais à l'appel suivant, on devrait pouvoir ne calculer que les images des éléments nouvellement ajoutés à `l0`. La méthode suivante évite ce travers :

```
let pt_fixe f l0 =
    let rec pt_fixe_rec définitifs productifs épuisés =
         match productifs with
              [] -> définitifs
            | a :: q -> if mem a épuisés then pt_fixe_rec définitifs q épuisés
                        else let l = f(a)
                             in
                             pt_fixe_rec (union l définitifs)
                                         (union l productifs)
                                         (a :: épuisés)
    in
    pt_fixe_rec l0 l0 [] ;;
```

On a désigné par `productifs` la liste des éléments dont on doit calculer les images, par `définitifs` la liste des images qui grossit petit à petit, et par `épuisés` la liste des éléments dont les images ont déjà été calculées, et adjointes à la liste des `définitifs`.

À vous, maintenant : cherchez un éventuel meilleur algorithme, et, surtout, faites-le moi parvenir, que je le répercute, et que nous en profitions tous.

Expressions régulières et automates

Nous continuons ici ce qui a été entamé dans la lettre numéro 0.
Rappelons que nous avions écrit un parseur pour les expressions régulières, dont le type était défini ainsi :

```
type expression_régulière =
      Chaîne of string
    | Caractère of char
    | Séquence of expression_régulière * expression_régulière
    | Étoilée of expression_régulière
    | Alternative of expression_régulière * expression_régulière ;;
```

En fait, nous nous étions arrangés (cf. la fonction `agrège`) pour qu'une expression régulière ne soit pas de la forme `Caractère(c)`.
Nous allons ici définir une représentation Caml des automates finis, et construire la fonction `auto_de_regexp` qui renvoie un automate quand on lui fournit une expression régulière.

Une implémentation des automates

Nous nous intéressons aux automates finis, déterministes ou non-déterministes. Dans ce dernier cas, nous autoriserons les ε-transitions, même si nous devrons écrire plus tard une fonction qui les supprime...

Le typage des automates

Programme 9 Le typage des automates déterministes

```
exception Pas_de_transition ;;

type genre      = Non_final | Final ;

type automateD  = { mutable état_initialD : étatD ;
                    mutable les_étatsD : étatD list }
and étatD       = { mutable espèceD : genre ;
                    mutable déclencheursD : char list ;
                    mutable transitionD : char -> étatD } ;;
```

Un automate déterministe est défini par la donnée de son état initial et de la liste de ses états. Un état d'un automate déterministe sera constitué de la liste des caractères susceptibles de provoquer une transition et d'une fonction de transition qui à un caractère associe l'état correspondant, ou renvoie une exception si le caractère fourni ne correspond à aucune transition de l'état vers un autre. En outre, nous spécifierons pour chaque état s'il est ou non final. On pourrait *a priori* se dispenser de préciser la liste des caractères qui provoquent une transition, dans la mesure où la fonction de transition déclenche alors une exception. De fait, il est plus pratique pour la suite que l'on ait un accès rapide à cette liste de caractères.

Pour les automates non déterministes, la fonction de transition ne renvoie plus un seul état mais une liste d'états ; en outre on fournit la liste (éventuellement vide) des états auxquels on accède par une ε-transition. Ceci se traduit ainsi :

Programme 10 Le typage des automates non déterministes

```
type automateND = { mutable état_initialND : étatND ;
                    mutable les_étatsND : étatND list }
and étatND      = { mutable espèceND : genre ;
                    mutable déclencheursND : char list ;
                    mutable transitionND : char -> étatND list ;
                    mutable epsilon_transitionND : étatND list } ;;
```

On notera qu'il y a bien d'autres façons de procéder…

Mon automate accepte-t-il ou non une chaîne ?

Il est assez facile d'écrire alors la fonction

`reconnaissanceD : automateD -> string -> bool`

qui dit si oui ou non une chaîne est reconnue par un automate fini déterministe (voir le programme 11, page suivante).

Programme 11 La reconnaissance par un automate déterministe

```
let reconnaissanceD automate chaîne =
    let n = string_length chaîne
    in
    let rec reconnaît_rec état i =
        if i = n then état.espèceD = Final
        else try reconnaît_rec (état.transitionD chaîne.[i]) (i+1)
             with Pas_de_transition -> false
    in
    reconnaît_rec automate.état_initialD 0 ;;
```

Le cas d'un automate non déterministe est déjà plus compliqué. Nous n'aurons en réalité pas l'usage de la fonction de reconnaissance associée à un tel automate, mais, parce que c'est un exercice amusant, nous l'écrivons quand même.
Au lieu d'aller d'état en état au fur et à mesure du parcours de la chaîne, nous construisons la liste des états auxquels on a pu accéder : à la fin du parcours, il suffira de voir si l'un de ces états est final.
Nous aurons pour cela besoin de calculer pour chaque état son ε-clôture, c'est-à-dire la liste des états auxquels des ε-transitions successives peuvent nous mener. C'est à chacun de ces états qu'on essaiera d'appliquer la transition provoquée par le caractère courant. Il s'agit du problème de point fixe dont nous avons déjà parlé… On trouvera le programme Caml correspondant ci-dessous (programme 12).

La génération des automates

Il nous reste à écrire une fonction `auto_de_regexp` qui prend une expression régulière, et renvoie un automate non déterministe qui la reconnaît. L'automate produit sera énorme et bourré d'ε-transitions. Nous devrions le simplifier et le rendre déterministe.

Programme 12 La reconnaissance par un automate non déterministe

```
let pt_fixe f l0 =
    let rec pt_fixe_rec défi prod épuisés =
         match prod with
              [] -> défi
            | a :: q -> if mem a épuisés then pt_fixe_rec défi q épuisés
                        else let l = f(a)
                             in
                             pt_fixe_rec (union l défi)
                                         (union l prod)
                                         (a :: épuisés)
    in
    pt_fixe_rec l0 l0 [] ;;

let epsilon_clôture liste_états =
        pt_fixe (function état -> état.epsilon_transitionND) liste_états ;;

let flat_union_map f l = flat_union_map_rec l where rec
          flat_union_map_rec = function
                | [] -> []
                | a :: q -> union (f a) (flat_union_map_rec q) ;;

let reconnaissanceND automate chaîne =
    let n = string_length chaîne
    in
    let rec reconnaîtND_rec liste_états i =
        if i = n then
            exists (function état -> état.espèceND = Final) liste_états
        else    let c = chaîne.[i]
                in
                reconnaîtND_rec
                    (flat_union_map
                        ( function état ->  try état.transitionND c
                                            with Pas_de_transition -> [] )
                        (epsilon_clôture liste_états) )
                    (i+1)
    in
    reconnaîtND_rec [ automate.état_initialND ] 0 ;;
```

L'alternative

Si A_1 et A_2 sont les automates respectivement associés aux expressions régulières e_1 et e_2, l'automate A associé à l'expression régulière $e = e_1 \mid e_2$ est construit ainsi : son état initial mène par ε-transitions aux états initiaux de A_1 et A_2 ; on ajoute une ε-transition de chacun des états finals de A_1 et A_2 (qui ne seront plus finals dans A) au nouvel état final.
On a donc besoin pour commencer de trouver la liste des états finaux d'un automate, ce que réalise la fonction `états_finaux`, en lignes 1–8 du programme 13.

Programme 13 Recherche des états finaux et gestion de l'alternative

```
1 let filtre prédicat = filtre_rec where rec
2     filtre_rec = function
3         | [] -> []
4         | a :: q -> if prédicat a then a :: (filtre_rec q)
5                     else filtre_rec q ;;
6
7 let états_finaux automate =
8     filtre (function état -> état.espèceND = Final) automate.les_étatsND ;;
9
10 let automate_d'alternative a1 a2 =
11     let finaux1,finaux2 = (états_finaux a1),(états_finaux a2)
12     in
13     let nouvel_initial =
14         { espèceND = Non_final ; déclencheursND = [] ;
15           epsilon_transitionND = [ a1.état_initialND ; a2.état_initialND ] ;
16           transitionND = (function _ -> raise Pas_de_transition)  }
17     and nouveau_final =
18         { espèceND = Final ; déclencheursND = [] ;
19           epsilon_transitionND = [] ;
20           transitionND = (function _ -> raise Pas_de_transition ) }
21     in
22     let foo état =
23         begin
24             état.espèceND <- Non_final ;
25             état.epsilon_transitionND <-
26                 nouveau_final :: état.epsilon_transitionND
27         end
28     in
29     do_list foo finaux1 ; do_list foo finaux2 ;
30
31     { état_initialND = nouvel_initial ;
32       les_étatsND = nouvel_initial :: nouveau_final
33           :: (a1.les_étatsND @ a2.les_étatsND) } ;;
```

L'étoile

Avec les notations précédentes, il est un peu plus difficile de construire l'automate A associé à l'expression régulière $e = e_1^*$. On créera un nouvel état initial pour A, duquel on arrive par des ε-transitions aussi bien à l'état initial de A_1 qu'au nouvel état final de A. En outre, de chaque état final de A_1 on pourra aller par une ε-transition sur l'état final de A mais aussi sur l'état initial de A_1. Cela donne le programme 14.

Programme 14 L'étoile

```
let automate_d'étoilée a1 =
    let finaux1 = états_finaux a1
    in
    let nouveau_final =
        {   espèceND = Final ; déclencheursND = [] ;
            epsilon_transitionND = [] ;
            transitionND = (function _ -> raise Pas_de_transition ) }
    in
    let nouvel_initial =
        {   espèceND = Non_final ; déclencheursND = [] ;
            epsilon_transitionND = [ a1.état_initialND ; nouveau_final ] ;
            transitionND = (function _ -> raise Pas_de_transition)  }
    in
    let foo état =
        begin
            état.espèceND <- Non_final ;
            état.epsilon_transitionND <-
                a1.état_initialND :: nouveau_final
                    :: état.epsilon_transitionND
        end
    in
    do_list foo finaux1 ;

    { état_initialND = nouvel_initial ;
      les_étatsND = nouvel_initial :: nouveau_final :: a1.les_étatsND } ;;
```

La séquence

Avec les notations précédentes, il est aisé de construire l'automate A associé à l'expression régulière $e = e_1 e_2$: il suffit d'ajouter des ε-transitions de chacun des états finals de A_1 vers l'état initial de A_2.

Les chaînes de caractères

L'automate de reconnaissance d'une chaîne ne présente vraiment pas la moindre difficulté.

On recolle les morceaux...

Il n'y a plus qu'à se laisser guider par la structure de l'expression régulière pour appliquer l'une des fonctions précédentes à bon escient.

Programme 15 La séquence

```
let automate_de_séquence a1 a2 =
    let finaux1 = états_finaux a1
    in
    let foo état =
        begin
            état.espèceND <- Non_final ;
            état.epsilon_transitionND <-
                a2.état_initialND :: état.epsilon_transitionND
        end
    in
    do_list foo finaux1 ;
    { état_initialND = a1.état_initialND ;
      les_étatsND = a1.les_étatsND @ a2.les_étatsND } ;;
```

Programme 16 La reconnaissance d'une chaîne

```
let automate_de_chaîne s =
    let final =
        {   espèceND = Final ; déclencheursND = [] ;
            epsilon_transitionND = [] ;
            transitionND = (function _ -> raise Pas_de_transition ) }
    in
    let rec ajoute_caractère i liste_des_états état_courant =
        if i >= 0 then
            let nouvel_état =
                { espèceND = Non_final ; déclencheursND = [ s.[i] ] ;
                  epsilon_transitionND = [] ;
                  transitionND =
                    (function c -> if c = s.[i] then [ état_courant ]
                                   else raise Pas_de_transition )     }
            in ajoute_caractère (i-1)
                                (nouvel_état::liste_des_états)
                                nouvel_état
        else
            { état_initialND = état_courant ;
              les_étatsND = liste_des_états }
    in
    ajoute_caractère ((string_length s) - 1) [ final ] final ;;
```

Programme 17 Automate associé à une expression régulière

```
let rec auto_de_regexp = function
      Chaîne s -> automate_de_chaîne s
    | Séquence(e1,e2) -> automate_de_séquence (auto_de_regexp e1) (auto_de_regexp e2)
    | Étoilée e -> automate_d'étoilée (auto_de_regexp e)
    | Alternative(e1,e2) -> automate_d'alternative (auto_de_regexp e1) (auto_de_regexp e2)
    | _ -> failwith "êtes-vous sûr d'avoir appelé le bon parseur ? " ;;
```

Pour conclure. . .

Nous ne continuerons pas davantage cette implémentation des automates : il resterait à écrire la suppression des ε-transitions, la déterminisation, et la minimisation de l'automate final. Tout cela est très bien fait dans notre bible commune : *Le langage Caml* de P. Weis et X. Leroy. On y trouvera une définition du type des automates très voisine de celle que nous avons utilisée ici. Je souhaitais surtout développer sur l'exemple des expressions régulières l'utilisation des flots, et réfléchir avec vous au problème du point fixe, qui m'a paru intéressant.