La lettre de Caml
numéro 0

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. (1) 48 05 16 04
Fax (1) 48 07 80 18

septembre 1995

Édito

Voici donc le numéro 0 de La lettre de Caml.
Mon intention est de lancer un petit bulletin de liaison entre utilisateurs de Caml Light, en pensant tout particulièrement aux collègues des classes prépas chargés de l'enseignement de l'option informatique.
Ce sera ce que nous en ferons : je dis nous *car j'espère bien pouvoir bientôt compter sur vos contributions.* A priori, *j'y verrais bien des exemples de programmation, des algorithmes, des analyses d'algorithme, des remarques sur nos implémentations, des utilitaires...*
Je travaille sur Macintosh, et ce que j'écris s'en ressentira sûrement. Mais je ne suis pas sectaire : les PC-istes qui voudraient collaborer sont les très bienvenus.
J'espère vos réactions, qui m'encourageront à continuer. Et pour la distribution, il faudra bien en reparler un de ces jours (les timbres augmentent en octobre, à ce que je crois savoir...)
Je suis réfractaire à toutes les bibles : le programme officiel de nos classes ne saurait être autre chose qu'un guide (un prétexte, même) pour ce bulletin ; nous nous évaderons sans scrupule.

Table des matières

La release 0.7 de Caml Light est disponible

Le serveur de l'INRIA (`ftp.inria.fr`) propose la nouvelle version de Caml Light, accompagnée de ses habituels dossiers d'exemples, de documentation et de sources. Quoi de neuf? quelques sucres syntaxiques, de nouvelles bibliothèques, et des plus pour les motifs de filtrage. L'interface est quelque peu améliorée grâce à l'introduction d'un *pretty-print*. Nous n'évoquerons pas ici quelques modifications bien plus subtiles (dans la gestion des types mutables, en particulier) qui sont de toutes façons transparentes pour les utilisateurs standard, comme moi. On lira avec intérêt le nouveau manuel de référence, disponible en versions texte, dvi et postscript sur le serveur de l'INRIA, et on trouvera aussi une mouture récente du tutorial de Michel Mauny.

Filtrage avec surveillants

Dans sa nouvelle version, Caml nous propose un supplément pour les filtrages, qu'ils se trouvent dans un `fun`, un `function`, un `match`, ou même un `try`. Si *`pattern`* est un motif de filtrage qui filtre un ensemble E d'expressions, et si *`cond`* est une expression qui s'évalue en un booléen, *`pattern`* `when` *`cond`* est un nouveau motif de filtrage qui ne retient des expressions de E que celles qui satisfont l'expression *`cond`*. Plus précisément, si le filtrage induit par *`pattern`* réussit, on évalue — avec les liaisons éventuellement ajoutées par le filtrage — l'expression *`cond`*. Si l'on obtient `true`, c'est que le motif complet filtre, sinon le motif est rejeté, et on passe, bien entendu, aux motifs suivants éventuels.
Par exemple, le filtrage suivant peut se rencontrer dans une recherche dichotomique dans un arbre :

```
let rec recherche_dichotomique x = function
    | Arbre(_,u,_) as a when u = x -> a
    | Arbre(gauche,u,_) when u > x -> recherche_dichotomique x gauche
    | Arbre(_,u,droit) when u < x -> recherche_dichotomique x droit
    | _ -> raise Not_found ;;
```

Caml parle de *guards*, ce que j'ai traduit par *surveillants*, mais il y a sûrement mieux. Notons au passage un petit détail cosmétique : Caml tolère maintenant la barre `|` superflue en tête du premier motif d'un filtrage, sans doute pour le joli aspect du filtrage ainsi complètement bordé d'un trait vertical. De la même façon, il tolère également les points-virgules `;` superflus devant `end` ou devant une accolade `}`, le tout étant bien sûr affaire de goût.

Quelques sucres

Au fait, savez-vous à qui l'on doit l'expression habituelle *sucre syntaxique* (en anglais, *of course*, *syntaxic sugar*)?
En plus des orthographes de la version 0.6, à savoir `&` et `or`, nous avons maintenant le droit de noter `&&` et `||` les connecteurs logiques ET et OU. Tout cela rappelle furieusement C, et a surtout l'avantage d'être un peu plus cohérent. Rappelons qu'il n'aurait pas été possible d'utiliser `or` et `and`, puisque `and` est réservé pour d'autres usages.

Les concepteurs de Caml ont imaginé deux nouvelles syntaxes pour gérer les chaînes de caractères — ce qui se révèlera sans doute plus pratique à l'usage — à savoir :

— au lieu d'écrire `nth_char s i`, nous pouvons écrire `s.[i]`, ce qui me semble plus pratique ;

— de même, au lieu de `set_nth_char s i c`, nous écrirons plus simplement `s.[i] <- c`.

À explorer encore

Caml fournit dans sa dernière version une nouvelle bibliothèque, `format`, qui permet d'écrire des *pretty-printer*, c'est-à-dire de gérer sauts de lignes, tabulations, ou encore *wrapping* (qui trouvera une traduction française correcte ? il s'agit de ce qui se passe quand, ne trouvant plus assez de place sur la ligne en cours, un éditeur de texte renvoie automatiquement le mot trop long au début de la ligne suivante). C'est sur la base de cette bibliothèque qu'a été conçue la partie de l'environnement Caml Light qui affiche les résultats successifs des évaluations. On obtient quelque chose d'assez plaisant pour des types complexes, ou des structures imbriquées.
Enfin, c'est en utilisant cette bibliothèque qu'on doit écrire, si on le souhaite, ses nouvelles fonctions d'impression, pour des types exotiques. Caml fournit en effet la fonction `install_printer : string -> unit` qui fonctionne ainsi : après `install_printer "ma_fonction" ;;`, où `ma_fonction` est du type `mon_type -> unit`, l'affichage de tout objet de type `mon_type` sera effectué par Caml en appelant `ma_fonction` sur l'objet en question. Il s'agit en quelque sorte d'un aiguillage de l'affichage par le type des objets. Je pense que cela peut permettre de jolies choses sur des structures comme des nombres complexes (qu'on écrira ainsi de la façon habituelle)...
Pourquoi ne pas se donner rendez-vous ici pour partager nos prochaines fonctions d'affichage ?

Lecture des expressions régulières

Introduction

Mon projet est ici de développer petit à petit un programme de reconnaissance des expressions régulières, en construisant les automates déterministes minimaux correspondants. Cela fera l'objet de plusieurs numéros de cette Lettre de Caml.
Pour aujourd'hui, on se contentera de lire et déchiffrer les expressions régulières. Il s'agit en fait d'une démarche tout à fait classique, systématisée complètement dans les ouvrages classiques de Aho, Hopcroft et Ullman, par exemple (je les cite car ils sont excellents).

Rappels sur les flots

Même si nous n'en parlerons sans doute pas à nos élèves, l'usage des flots (*streams* en anglais) est incontournable dès qu'il s'agit d'écrire des analyseurs lexicaux (j'utilise le néologisme *lexeur*) ou syntaxiques (et le néologisme *parseur*). Notons à ce propos que le texte de Mauny, excellent par ailleurs, est plutôt confus et compliqué sur le sujet du filtrage des flots.

Rappelons que les flots sont créés le plus souvent à l'aide des deux fonctions `stream_of_string`, pour lire les caractères éléments du flot depuis une chaîne, et `stream_of_channel`, pour lire les éléments du flot depuis un fichier. On peut cependant aussi les créer "à la main" (voir, pour le *fun*, plus loin, ce qui est fait sur le crible d'Ératosthène).

L'accès à un flot est destructif. Si je lis le premier item d'un flot, je le retire du flot. Ainsi n'y a-t-il pas dans le filtrage d'un flot de notion de queue (de `cdr`, si vous préférez), comme pour les listes. Si le filtrage habituel sur les listes s'écrit

```
match liste with
    | a :: q -> ...
    | [] -> ...
```

le filtrage habituel sur les flots est plutôt du genre

```
match flot with
    | [< 'x >] -> ...
    | [< 'y >] -> ...
    | [< >] ->
```

Au lieu d'appliquer récursivement la fonction sur `q` dans le cas de la liste, on l'appliquera au flot lui-même, puisque le filtrage en aura supprimé la tête.

Rappelons brièvement la syntaxe des flots : `[< 'x ; f1 ; 'y >]` représente un flot constitué d'un premier élément `x`, d'un sous-flot `f1`, et d'un élément `y`. Si le flot est de type `'a stream`, alors `x` et `y` sont du type `'a`, et, bien sûr, `f1` du type `'a stream`. L'erreur la plus fréquente est sans doute d'oublier l'accent grave (*quote* : `'`) devant les éléments du flot.

Le filtrage se passe de façon tout à fait particulière. Le plus surprenant est sans doute que le filtrage ne choisit le motif filtrant qu'à l'aide du seul premier élément de chaque motif. Ainsi, par exemple, Caml signalera que, dans le cas suivant, le deuxième motif n'a aucune chance d'être utilisé :

```
>       Caml Light version 0.7mac

#let bidon flot = match flot with
    | [< '1 ; '2 ; s >] -> s
    | [< '1 ; '3 ; s >] -> s
    | [< '2 ; s >] -> s
    | [< >] -> failwith "non prévu" ;;
Toplevel input:
>    | [< '1 ; '3 ; s >] -> s
>         ^
Warning: this matching case is unused.
bidon : int stream -> int stream = <fun>
```

Remarquons que le premier cas de filtrage aurait pu tout aussi bien s'écrire `[< '1 ; '2 >] -> flot` puisque les deux premiers éléments auront alors été

détruits du flot. Le dernier cas du filtrage avale un flot vide, donc n'importe quel flot, puisque tout flot commence par du vide...
Dans le cas où le filtrage est aiguillé sur un des motifs, il ne peut changer d'avis ; si la suite du motif ne passe pas, on déclenche l'exception `Parse_error`, et ainsi on obtiendra en poursuivant l'exemple précédent :

```
#bidon [< '1 ; '4 >] ;;
Uncaught exception: Parse_error
#bidon [< '1 ; '3 >] ;;
Uncaught exception: Parse_error
```

En revanche, si aucun (début de) motif ne filtre le (début du) flot, c'est l'exception `Parse_failure` qui est déclenchée. Ainsi :

```
#let autre_bidon flot = match flot with
   | [< '1 >] -> "oui"
   | [< '2 >] -> "non" ;;
autre_bidon : int stream -> string = <fun>
#autre_bidon [< '2 >] ;;
- : string = "non"
#autre_bidon [< '5 >] ;;
Uncaught exception: Parse_failure
```

Il y a beaucoup plus rusé (on peut discuter de la correction de la syntaxe choisie — m'enfin, y a-t-il mieux ?)... on verra plus loin que cette nouvelle façon de filtrer un flot est précieuse.
Si `f` est une fonction du type `'a stream -> 'b`, et si `flot` est du type `'a stream`, le motif `[< f x >]` filtre tout flot de type `'a stream`, et crée une nouvelle liaison, qui à la variable `x` lie la valeur de la fonction `f` sur le début du flot. Un exemple rend sans doute les choses plus claires.

```
#let int_char c = (int_of_char c) - (int_of_char `0`) ;;
int_char : char -> int = <fun>
#let rec mange_int accu flot = match flot with
   | [< '(`0`..`9`) as x >] -> mange_int (10*accu + (int_char x)) flot
   | [< >] -> accu ;;
mange_int : int -> char stream -> int = <fun>
#let mange_entier = mange_int 0 ;;
mange_entier : char stream -> int = <fun>
#let mange_plus flot = match flot with
    [< mange_entier n1 ; ' `+` ; mange_entier n2 >] -> n1 + n2 ;;
mange_plus : char stream -> int = <fun>
#mange_plus (stream_of_string "123+456") ;;
- : int = 579
```

Nous en savons assez pour passer à l'écriture du...

Lexeur

Il n'est pas trop difficile d'écrire un lexeur d'expressions régulières : tout caractère est significatif, les caractères spéciaux sont les deux parenthèses (pour grouper), la barre verticale (pour le choix), l'astérisque (pour l'étoilement). Bien sûr, on ajoute un caractère pour avaler les caractères spéciaux, et évidemment on a choisi comme tout le monde le backslash. Notons que les espaces

sont réputés significatifs. Notons aussi que l'on pourrait apprendre au lexeur des conventions telles que `\t` pour la tabulation ; cela ne présente pas la moindre difficulté.
Voici le fichier-source du lexeur obtenu.

Programme 1 Un lexeur d'expressions régulières

```
let string_of_char = make_string 1 ;;

type lexeme =
    Car of char | Étoile | Parenthèse_gauche | Parenthèse_droite | Barre ;;

let rec lexeur flot = match flot with
      [< '`\\` ; 'x >] -> [< '(Car x) ; lexeur flot >]
    | [< '`|` >] -> [< 'Barre ; lexeur flot >]
    | [< '`*` >] -> [< 'Étoile ; lexeur flot >]
    | [< '`(` >] -> [< 'Parenthèse_gauche ; lexeur flot >]
    | [< '`)` >] -> [< 'Parenthèse_droite ; lexeur flot >]
    | [< 'x >] -> [< '(Car x) ; lexeur flot >]
    | [< >] -> [< >] ;;
```

Parseur

Passons maintenant à l'analyse syntaxique de l'expression régulière.
Rappelons notre grammaire initiale des expressions régulières :

```
ExprReg ::=   Caractère
            | (ExprReg)
            | ExprReg ExprReg
            | ExprReg | ExprReg
            | ExprReg *
```

Des problèmes de grammaires

Est-il besoin de le faire observer ? notre grammaire est ambigue.
La parade est classique : on utilise les priorités habituelles des opérateurs. À savoir dans le cas qui nous intéresse, et dans l'ordre, l'étoile, la concaténation, et le choix. On écrit ainsi, avec les méthodes classiques développées dans les Aho, Hopcroft et Ullman déjà cités, la grammaire non ambigüe suivante :

```
E ::= F | E
    | F
F ::= G F
    | G
G ::= H *
    | H
H ::= caractère
    | ( E )
```

Il y a encore un problème, puisque le filtrage des flots ne se détermine que sur le premier élément : il faut encore factoriser cette grammaire à gauche. Ce qui donne :

```
E   ::=  F E1
E1  ::=  |E
       | ∅
F   ::=  G F1
F1  ::=  F
       | ∅
G   ::= H G1
G1  ::= *
      | ∅
H   ::= caractère
      | ( E )
```

Écriture du parseur

Le miracle, c'est qu'une fois la grammaire non ambigue et factorisée à gauche écrite, il n'y a plus rien à faire. On traduit mot à mot les différentes clauses de la grammaire, ce qui est fait dans le programme page suivante.
Simplement, on a ajouté une procédure `agrège` (dans l'intention de simplifier la construction des automates correspondants) qui remplace une concaténation de caractères par une chaîne : une expression aussi banale que `(abc)*|a*|(ab)*c` en sortira grandement simplifiée.

À suivre. . .

Nous verrons dans les prochains numéros comment implémenter les algorithmes bien connus de déterminisation, de suppression des ε-transitions et de minimisation des automates finis. Combinant le tout, nous pourrons écrire une fonction qui, étant donnée la chaîne de caractères représentant une expression régulière, fournit l'automate déterministe minimal qui lui correspond, ou encore la fonction de reconnaissance elle-même, qui renvoie sur une chaîne quelconque `true` ou `false` selon qu'elle fait partie ou non du langage rationnel décrit par l'expression régulière considérée.

Programme 2 Un parseur d'expressions régulières

```
include "lexeur_expression_régulière" ;;

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 ;;

let rec parse_E flot = match flot with
    [< parse_F f ; parse_E1 e1 >] -> match e1 with
              [< 'expr >] -> Alternative(f,expr)
            | [< >] -> f
and parse_E1 flot = match flot with
      [< 'Barre ; parse_E e >] -> [< 'e >]
    | [< >] -> [< >]
and parse_F flot = match flot with
    [< parse_G g ; parse_F1 f1 >] -> match f1 with
              [< 'expr >] -> Séquence(g,expr)
            | [< >] -> g
and parse_F1 flot = match flot with
      [< parse_F f >] -> [< 'f >]
    | [< >] -> [< >]
and parse_G flot = match flot with
    [< parse_H h ; parse_G1 g1 >] -> match g1 with
              [< 'bidon >] -> Étoilée(h)
            | [< >] -> h
and parse_G1 flot = match flot with
      [< 'Étoile >] -> [< 'Caractère(`? `) >]
    | [< >] -> [< >]
and parse_H flot = match flot with
      [< '(Car c) >] -> Caractère(c)
    | [< 'Parenthèse_gauche ; parse_E expr ; 'Parenthèse_droite >] -> expr
;;

let rec agrège = function
      Séquence(e1,e2)
            ->  ( match (agrège e1),(agrège e2) with
                        (Chaîne s1),(Chaîne s2) -> Chaîne (s1 ^ s2)
                      | (Chaîne s1),(Séquence ((Chaîne s2),e))
                          -> agrège (Séquence (Chaîne (s1^s2),e))
                      | (Séquence (e,(Chaîne s1))),(Chaîne s2)
                          -> agrège (Séquence (e,Chaîne(s1^s2)))
                      | e'1,e'2 -> Séquence(e'1,e'2)
                )
    | Étoilée(e1) -> Étoilée(agrège e1)
    | Alternative(e1,e2) -> Alternative((agrège e1),(agrège e2))
    | Caractère(c) -> Chaîne (string_of_char c)
    | e -> e ;;

let parseur s = agrège (parse_E (lexeur (stream_of_string s))) ;;
```

Les flots pervertis : le crible d'Ératosthène

L'écriture des analyseurs (lexicaux ou syntaxiques) ne fait pas le seul intérêt des flots. En effet, un autre point de vue est possible, et riche d'implications.
Les flots constituent en effet les seuls objets de Caml qui soient soumis à ce qu'on appelle traditionnellement l'évaluation paresseuse (en anglais *lazy evaluation*), puisque les éléments d'un flot, outre qu'ils sont retirés du flot au fur et à mesure qu'on les lit, sont évalués à ce moment là seulement. On écrit sans vergogne des procédures récursives sur les flots, sans se préoccuper de leur arrêt. En fait les évaluations ne sont faites qu'à la demande, c'est-à-dire uniquement si l'on tente d'accéder aux éléments du flot.
C'est ainsi qu'on écrira facilement un crible d'Ératosthène à l'aide d'un flot d'entiers. Je vous laisse regarder le programme suivant. `list_and_stream` lit les premiers éléments d'un flot dont elle fait une liste, et renvoie également le flot restant.

Programme 3 Le crible d'Ératosthène

```
let rec à_partir_de n = [< 'n ; (à_partir_de (n+1)) >] ;;

let rec filtre_stream f flot = match flot with
    | [< 'x >] ->   if f(x) then [< 'x ; (filtre_stream f flot) >]
                    else [< (filtre_stream f flot) >]
    | [< >] -> [< >] ;;

let ne_divise_pas a b = (b mod a) <> 0 ;;

let rec crible flot = match flot with
    | [< 'n >] ->   [<
                        'n ;
                        (crible (filtre_stream (ne_divise_pas n) flot))
                    >]
    | [< >] -> [< >] ;;

let nombres_premiers = crible (à_partir_de 2) ;;

let rec list_and_stream n flot =
    if n = 0 then [] , flot
    else match flot with [< 'x >]
            ->  let l,f = list_and_stream (n-1) flot
                in
                (x :: l) , f ;;

let list_of_stream n flot = fst (list_and_stream n flot) ;;
```

Et voilà ce que cela donne à l'usage :

```
>       Caml Light version 0.7mac

include "flots.crible.ml";;
à_partir_de : int -> int stream = <fun>
filtre_stream : ('a -> bool) -> 'a stream -> 'a stream = <fun>
ne_divise_pas : int -> int -> bool = <fun>
crible : int stream -> int stream = <fun>
```

```
nombres_premiers : int stream = <abstr>
list_and_stream : int -> 'a stream -> 'a list * 'a stream = <fun>
list_of_stream : int -> 'a stream -> 'a list = <fun>
- : unit = ()
#list_of_stream 30 nombres_premiers ;;
- : int list =
 [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71;
  73; 79; 83; 89; 97; 101; 103; 107; 109; 113]
```

En réalité, il y a moyen de simuler le même processus sans utiliser de flots, c'est ce qui est fait dans le programme de la page suivante, mais c'est beaucoup plus compliqué...

Caml et édition de texte

Pour les utilisateurs de Macintosh seulement : BBEdit

Je ne sais pas comment vous palliez l'indigence de l'éditeur de la fenêtre `input` de Caml Light. Pour moi, j'écris tout programme de plus de 10 lignes dans un éditeur séparé, puis je procède ou bien par copier-coller, ou plus souvent par *include*.
Mon éditeur favori est `BBEdit`, de *Bare Bones Software*. Je peux vous distribuer gratuitement (contre une disquette) la version de démonstration, qui est déjà très complète et convaincante. En revanche, je pense que nous pouvons tous débourser les quelques dollars que réclament *Bare Bones Software* pour la version complète (sur CD-Rom ! , avec plein d'utilitaires...), d'autant qu'ils paraissent assez sympas, et m'ont renvoyé une nouvelle version gratuitement, sans que je leur demande rien...
Cet éditeur a de nombreux atouts : un *browser* bien fait qui remplace avantageusement le Finder ; la possibilité de lire et créer des fichiers HTML ; un accès direct à et une collaboration avec MPW, Symantec, CodeWarrior (si vous les possédez) ; un interfaçage avec Think Reference et Toolbox Assistant ; un interfaçage avec Kodex et Internet Config ; etc.
Plus simplement, il connaît les modes Unix, Dos et Mac pour la lecture et l'écriture des fichiers texte ; il sait faire les conversions tabulations/espaces, afficher les caractères étranges (gremlins) ou les filtrer ; il possède un mode de recherche-remplacement puissant, avec un `grep` complet et une recherche multi-fichiers.
Et surtout, il est programmable, au prix de quelques lignes de C.

Mes petites extensions

J'ai écrit deux extensions — que je tiens gracieusement à votre disposition, bien sûr — pour BBEdit, qui m'aident bien quand je programme en Caml. En effet, quiconque a fait quelques *include* a dû être confronté à des erreurs (non, pas vous ?). Or Caml indique alors dans un message agrémenté de jolis caractères spéciaux un intervalle d'entiers, comme $[425, 493]$, indiquant par là que l'erreur est dans le fichier indiqué, du caractère 425 au caractère 493. L'extension

Programme 4 Crible d'Ératosthène sans les flots

```
type 'a suite_infinie = Nil | Cellule of (unit -> 'a * 'a suite_infinie) ;;

exception Suite_Vide ;;

let cons x l =
    let f () = (x,l)
    in Cellule f  ;;

let tête = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,_ -> x ;;

let queue = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with _,q -> q ;;

let est_vide = function
      Nil -> true
    | _ -> false ;;

let rec force n l = match n,l with
      0,l -> [],l
    | n,Nil -> raise Suite_Vide
    | n,Cellule f ->
            match f() with x,q ->   let liste,reste = force (n-1) q
                                    in x :: liste,reste ;;

let rec à_partir_de n = let f () = n,(à_partir_de (n+1)) in Cellule f ;;

let premiers n l = match force n l with liste,_ -> liste ;;
let reste n l = match force n l with _,r -> r ;;

let rec filtre prédicat = function
      Nil -> Nil
    | Cellule f -> match f() with x,q ->
        if (prédicat x) then
            let g () = x,(filtre prédicat q)
            in Cellule g
        else filtre prédicat q ;;

let non_multiple a b = (b mod a) <> 0 ;;

let élimine x l = filtre (non_multiple x) l ;;

let rec crible = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,q ->
            let g() = x,(crible (élimine x q))
            in Cellule g ;;

let nombres_premiers = crible (à_partir_de 2) ;;
```

`Intervalle` permet justement de sélectionner la zone de texte correspondante. Très simple, mais fort pratique.
D'autre part, j'ai toujours aimé utiliser des identificateurs de variables les plus parlants possible. On hésite souvent à faire ainsi car cela augmente parfois considérablement le travail de dactylographie, ce qui est fort dommage, car on y perd en lisibilité. C'est pour cela que j'ai écrit l'extension `Complétion` qui permet d'économiser la frappe.
Supposons qu'on ait écrit les quelques lignes de Caml suivantes :

```
let rec list_length = function
    | [] -> 0
    | a :: q -> 1 + (list_length q)
and list_integer i j =
    if i > j then []
    else i :: (l
```

et qu'on n'ait pas le courage de tout taper : `list_integer (i+1) j ;;`.
On se contentera d'appeler l'extension `Complétion` (on peut lui affecter un raccourci-clavier) après avoir tapé le *l*. Automatiquement apparaîtra ***let*** (je mets en gras ce qui est sélectionné), puisque *let* est le premier mot trouvé dans le fichier qui commence par le *l* initial. Bien sûr ce n'est pas ce qu'on veut. On rappelle `Complétion`. Cette fois on obtient *l**ist_length***, on recommence, et apparaît *l**ist_integer*** : en trois frappes clavier on a retrouvé le mot cherché. L'extension produit cycliquement tous les mots qui débutent par le préfixe fourni. À l'usage, ça m'est devenu tout simplement indispensable. (Pour les curieux, c'est Pierre Weis qui m'a donné l'idée d'écrire cette extension, car il a évoqué plusieurs fois lors du stage de Luminy de cette année cette fonctionnalité de son éditeur sous Unix — sans doute `emacs`.)

Aux utilisateurs de PC

Pouvez-vous m'indiquer comment vous procédez, et si vous avez développé des solutions analogues ?

À vos plumes...

J'attends avec impatience vos réactions, vos commentaires, voire vos engueulades. Et surtout, j'espère que quelques uns d'entre vous auront envie de participer à ce petit bulletin. Je me refuse à engager des fonds pour me connecter à Internet, et attends donc ce qui me paraît exigible : que l'administration nous ouvre des comptes. Alors seulement on pourra me joindre par ce biais ; en attendant, vous trouverez mes coordonnées complètes sur la première page.
À vous lire...