18. LE FILTRAGE (DEUXIEME PARTIE)

Au chapitre 14 nous avons construit une fonction de filtrage, la fonction **MATCH**, qui permet de comparer une liste avec un *modèle*, ou *filtre*, de liste.

Le filtre peut contenir 3 signes particuliers :

1. le signe **$** indique un élément quelconque,
2. le signe **&** indique une séquence, éventuellement vide, d'éléments quelconques,
3. le signe **%** indique un choix entre plusieurs éléments donnés.

Ainsi, le filtre **($ $ $)** correspond à toute liste comprenant exactement 3 éléments, le filtre **((% A B C) $)** à toute liste de exactement 2 éléments commençant par l'un des trois éléments **A**, **B** ou **C**, et le filtre **(DEBUT & FIN)** à toute liste commençant par l'élément **DEBUT** et se terminant par l'élément **FIN**.

Cette fonction de filtrage n'est pas complète : on est parfois non seulement intéressé par la structure de la liste, mais aussi par l'élément qui se situe à l'endroit d'un des signes particuliers. Ainsi, lorsqu'on compare le filtre **($ SUR A)** à un ensemble de données, on aimerait - peut-être - savoir ce qui se trouve à l'endroit du signe **$**.

Enrichissons alors un peu la syntaxe de nos filtres pour permettre que les opérateurs de filtrage puissent être suivis du nom d'une variable, tel qu'au lieu d'écrire :

($ SUR A)

nous puissions écrire :

($X SUR A)

De plus, enrichissons notre fonction de filtrage de manière à non seulement indiquer qu'une certaine donnée correspond au filtre, mais d'également indiquer quel élément se trouve à l'endroit de cette variable.

Par exemple, si nous filtrons

($X SUR A)

avec la donnée

(D SUR A)

la fonction de filtrage ramène **T**, c'est-à-dire : l'indication que la donnée correspond au filtre, mais aussi l'information qu'à l'endroit de **$X** se trouve l'élément **D**.

Evidemment, deux problèmes se posent. D'abord, il faut changer notre fonction **MATCH** de manière à ramener *deux* valeurs : l'indication de réussite ou d'échec du filtrage, plus éventuellement les *liaisons* des diverses variables du filtre. Ensuite il faut trouver un moyen d'exprimer ces liaisons. Deux possibilités existent : soit en le faisant comme LISP, c'est-à-dire en changeant les C-valeurs des variables (avec la fonction **SET** ou **SETQ**), soit en retournant une liste contenant textuellement les noms des variables suivis de leurs valeurs, donc une liste de la forme :

(($variable_1$ **.** $valeur_1$**) . . . (**$variable_n$ **.** $valeur_n$**))**

Des listes de cette forme s'appellent des *A-listes* ou des *listes d'associations*. Les toutes premières implémentations de LISP utilisaient de telles listes comme *environnement*, c'est-à-dire comme mémoire des liaisons actuelles et passées.

Ainsi, le résultat du filtrage présenté ci-dessus doit livrer le résultat :

(T ((X . D)))

où le **T** au début de la liste est l'indicateur de réussite du filtrage, et le deuxième élément de la liste est la A-liste exprimant la liaison de **X** à **D**.[1] Donnons, en guise de définition de nos nouveaux filtres quelques exemples de filtrage :

• (MATCH '(A B C D) '(A B C D)) → (T NIL)
Si le filtre ne contient aucun signe spécial le filtrage ne réussit qu'avec une donnée qui est **EQUAL** au filtre. Le résultat est une liste, contenant en **CAR** l'indicateur de réussite du filtrage **T** et contenant en **CADR** la A-liste vide, puisqu'aucune liaison n'est nécessaire.

• (MATCH '($X B C $-) '(A B C D)) → (T ((X . A)))
Au filtre commençant par le signe **$** doit correspondre un unique élément dans la donnée. Si le signe **$** est suivi d'un atome différent de **-**, cet atome est alors lié à l'élément correspondant dans la donnée. Le filtre spécial **$-** indique qu'un élément correspondant doit être présent dans la donnée, mais sans que l'on procède à une liaison. Ce dernier filtre correspond alors au signe **$** de notre fonction de filtrage du chapitre 14.

• (MATCH '($X $Y $Z $X) '(A B C A)) → (T ((X . A) (Y . B) (Z . C)))
Une répétition d'une même variable de filtre à l'intérieur d'un filtre indique des éléments égaux. Ainsi, au filtre ci-dessus correspond une donnée de 4 éléments dont le premier et le dernier élément sont identiques.

• (MATCH '($X &- $X) '(A B C A)) → (T ((X . A)))
Ce filtre décrit toute liste commençant et se terminant par le même élément. Le filtre **&-** correspond au filtre **&** de notre fonction **MATCH** du chapitre 14, il filtre donc un segment de longueur quelconque *sans* procéder à une liaison.

• (MATCH '($X &Y $X) '(A B C A)) → (T ((X . A) (Y B C)))
La différence entre ce filtre et le précédent est que le segment filtré est lié à la variable **Y**. Cette variable est donc liée à la liste **(B C)**. (Rappelons que le **CDR** de chaque doublet de la A-liste contient la valeur de la variable du **CAR** du doublet. C'est pourquoi nous avons le doublet **. . (Y B C) . .** dans la A-liste résultat.)

• (MATCH '(%(A B C) &-) '(A B C A)) → (T NIL)
Comme préalablement, le signe spécial **%** indique un choix multiple. Ce filtre décrit donc toute liste de longueur quelconque commençant soit par l'atome **A**, soit par **B**, soit par **C**.

Modifions alors notre programme de filtrage. Rappelons le :

```
(DE MATCH (FILTRE DONNEE) (COND
      ((ATOM FILTRE) (EQ FILTRE DONNEE))
```

[1] Si l'on désire une liaison *superficielle*, c'est-à-dire une liaison qui modifie les C-valeur des variables, il suffit de faire sur la partie A-liste du résultat :

(MAPC *A-liste* **(LAMBDA (X) (SET (CAR X) (CDR X))))**

```
((ATOM DONNEE) NIL)
((EQ (CAR FILTRE) '$)
     (MATCH (CDR FILTRE) (CDR DONNEE)))
((EQ (CAR FILTRE) '&) (COND
     ((MATCH (CDR FILTRE) DONNEE))
     (DONNEE (MATCH FILTRE (CDR DONNEE)))))
((AND (CONSP (CAR FILTRE))(EQ (CAAR FILTRE) '%))
     (LET ((AUX (CDR (CAR FILTRE)))) (COND
          ((NULL AUX) NIL)
          ((MATCH (CAR AUX) (CAR DONNEE))
               (MATCH (CDR FILTRE) (CDR DONNEE)))
          (T (SELF (CDR AUX))))))
((MATCH (CAR FILTRE) (CAR DONNEE))
     (MATCH (CDR FILTRE) (CDR DONNEE))))
```

En premier lieu, il s'agit d'introduire la A-liste. Changeons donc le début de la définition de **MATCH** en :

```
(DE MATCH (FILTRE DONNEE)
     (LET ((ALISTE NIL)) . . .
```

La taˆche sera divisée en deux : le travail proprement dit sera réalisé par une fonction auxiliaire, la fonction **MATCH** se contentant de lancer cette fonction auxiliaire et d'éditer le résultat.

```
(DE MATCH (FILTRE DONNEE)
     (LET ((ALISTE NIL))
          (IF (MATCH-AUX FILTRE DONNEE)
               (LIST T ALISTE) NIL)))
```

Ensuite réécrivons l'ancienne fonction **MATCH** qui s'appelle maintenant **MATCH-AUX** : dans les deux premières clauses il n'y a rien à changer. Les modifications doivent se situer dans le traitement des signes spéciaux **$** et **&**.

Résolvons d'abord la difficulté qui consiste à séparer le signe (**$** ou **&**) du nom de la variable qui suit. Si nous utilisons la fonction **EXPLODE** pour tester le premier caractère d'un atome, nous devons le faire avec *tous* les atomes contenus dans le filtre. C'est trop couˆteux en temps. Définissons donc les caractères **$**, **&** et **%** comme des macro-caractères. Ainsi, dès la lecture les signes spéciaux seront séparés des noms des variables qui suivent. Voici les définitions de ces trois caractères.

```
(DMC "$" () (CONS "$" (READ)))

(DMC "&" () (CONS "&" (READ)))

(DMC "%" () (CONS "%" (READ)))
```

Ainsi, le filtre :

```
($X &Y %(A B C))
```

sera traduit en :

```
(($ . X) (& . Y) (% A B C))
```

et les tests de présence de signes spéciaux n'a lieu que si le filtre débute par une sous-liste. Nous pouvons donc introduire une clause supplémentaire testant si le filtre commence par un atome. Si c'est le cas, nous savons qu'il doit s'agir d'une constante et qu'il suffit de comparer cette

constante avec le premier élément de la donnée pour ensuite relancer le processus de filtrage sur le reste des deux listes. Ce qui nous donne :

```
((ATOM (CAR FILTRE))
    (AND (EQ (CAR FILTRE) (CAR DONNEE))
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE))))
```

Dans tous les autres cas, le filtre débute par une sous-liste. Cette sous-liste peut alors correspondre à un des caractères spéciaux. Etudions alors tous les cas et commençons par le caractère **$** :

((EQUAL (CAAR FILTRE) "$") . . .)

Trois sous-cas peuvent se présenter :

1. la donnée est la liste vide. Dans ce cas le filtrage ne réussit pas et nous pouvons sortir en ramenant **NIL**.
2. le point d'exclamation est suivi du signe '-'. Nous nous trouvons alors dans le même cas que dans l'ancienne fonction **MATCH** : il ne reste qu'à comparer le reste du filtre avec le reste de la donnée.
3. le point d'exclamation est suivi du nom d'une variable. Deux cas sont alors à distinguer :
    1. Cette variable n'a pas encore de liaison sur la A-liste. Il faut alors lier la variable à l'élément correspondant dans la donnée et continuer à comparer le reste du filtre avec le reste de la donnée.
    2. La variable a déjà été liée sur la A-liste. Il faut alors comparer le filtre construit en remplaçant le premier élément par la valeur de la variable avec la donnée.

Construisons d'abord une petite fonction auxiliaire **ASSQ** qui cherche sur une A-liste si une variable donnée est déjà liée ou pas. Si elle est liée, **ASSQ** ramène le doublet **(***variable* **.** *valeur***)** :

```
(DE ASSQ (VARIABLE ALISTE) (COND
        ((NULL ALISTE) ())
        ((EQ (CAAR ALISTE) VARIABLE) (CAR ALISTE))
        (T (ASSQ VARIABLE (CDR ALISTE)))))
```

Cette fonction existe en tant que fonction standard dans la plupart des systèmes LISP.

Maintenant, nous pouvons écrire la clause pour le **$** :

```
((EQUAL (CAAR FILTRE) "$") (COND
    ((NULL DONNEE) ())
    ((EQ (CDAR FILTRE) '-)
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))
    ; regardons si la variable a une liaison ;
    ((ASSQ (CDAR FILTRE) ALISTE)
        (MATCH-AUX
            (CONS (CDR (ASSQ (CDAR FILTRE) ALISTE)) (CDR FILTRE))
            DONNEE))
    ; il n'y a pas de liaison ;
    (T (SETQ ALIST (CONS (CONS (CDAR FILTRE) (CAR DONNEE)) ALISTE))
        (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))))
```

Visiblement, cette clause reprend un à un chacun des cas présentés ci-dessus. Il subsiste une seule maladresse : le double calcul de la liaison de la variable. Il est possible d'attribuer une variable de plus à notre fonction **MATCH-AUX**, que nous nommerons **AUX** pour 'auxiliaire´, et d'utiliser cette variable pour sauvegarder temporairement la liaison. Ceci implique que nous modifions la fonction **MATCH-AUX** en :

(DE MATCH-AUX (FILTRE DONNEE)
(LET ((AUX NIL)) . . .

et récrivons l'avant dernière clause :

```
((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
      (MATCH-AUX (CONS (CDR AUX) (CDR FILTRE)) DONNEE))
```

Prenons ensuite le cas du filtre **%** de choix multiple explicite. La différence avec l'ancienne fonction réside dans la sauvegarde de l'état de la A-liste pendant les essais successifs; ce qui nous permet d'introduire des filtres à l'intérieur de la liste des choix. Ainsi, le filtre :

(%(DO RE (MI $-) (FA $- $-)) &-)

décrit toute liste commençant soit par l'atome **DO**, soit par l'atome **RE**, soit par une liste de deux éléments commençant avec **MI**, soit par une liste de trois éléments et débutant avec l'atome **FA**.

Voici donc la clause correspondant au signe **%** :

```
((EQUAL (CAAR FILTRE) "%")
    (LET ((AUX (CDAR FILTRE)) (ALIST2 ALIST)) (COND
          ((NULL AUX) (SETQ ALIST ALIST2) NIL)
          ((MATCH-AUX (CONS (NEXTL AUX) (CDR FILTRE)) DONNEE) T)
          (T (SELF (CDR AUX) ALIST2)))))
```

Vient ensuite le cas difficile des variables de segment, i.e.: des variables précédées du signe spécial '**&**'. Nous pouvons distinguer trois cas :

1. La variable est déjà liée. Il faut alors, dans le filtre, remplacer la variable par sa valeur et comparer le filtre ainsi obtenu avec la donnée.
2. le reste du filtre est vide. Il faut alors lier la variable (sauf si c'est le signe spécial '**-**') à la donnée et c'est terminé avec succès.
3. Sinon, il faut lier la variable (avec la même exception que ci-dessus) au segment vide **NIL**, regarder si le reste du filtre se superpose à la donnée, et si ça ne marche pas, il faut ajouter le premier élément de la donnée à la valeur de la variable, comparer le reste du filtre avec le reste de la donnée, et ainsi de suite jusqu'à obtenir un succès ou un échec définitif.

Ok. Allons-y et écrivons cette clause difficile :

```
((EQUAL (CAAR FILTRE) "&") (COND
   ; exite-il une valeur ;
   ((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (CDR AUX) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))
      ; succès ;
      (if (EQ (CDAR FILTRE) '-) ()
         (SETQ ALISTE (CONS (CONS (CDAR FILTRE) DONNEE) ALISTE)))
       T)
   (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()
         (SETQ ALISTE (CONS (LIST (CDAR FILTRE)) ALISTE)))
      (LET ((ALIST2 ALISTE))
           (IF DONNEE
              ; faisons les comparaisons successives ;
              (IF (MATCH-AUX (CDR FILTRE) DONNEE) T
                 (SETQ ALISTE ALIST2)
                 (IF AUX (NEXTL DONNEE)
                       ; la liaison de la variable s'agrandit ;
                       (NCONC (CAR ALISTE) (LIST (NEXTL DONNEE))))
                 (SELF ALISTE))
              ; échec totale : restaurons la A-liste ;
              (IF AUX (NEXTL ALIST))
              NIL)))))
```

Il ne reste que le dernier cas : le filtre commence par une sous-liste ordinaire. Comme dans l'ancienne version, il faut alors comparer le premier élément du filtre avec le premier élément de la donnée, et - si ça marche - il faut encore comparer le reste du filtre avec le reste des données.

Voici un ensemble de tests de cette nouvelle fonction **MATCH** :

(MATCH '(A B C) '(A B C))	→	**(T NIL)**
(MATCH '($- B C) '(A B C))	→	**(T NIL)**
(MATCH '($- B C $-) '(A B C))	→	**NIL**
(MATCH '($X B C) '(A B C))	→	**(T ((X . A)))**
(MATCH '($X B $Y) '(A B C))	→	**(T ((Y . C) (X . A)))**
(MATCH '($X B $X) '(A B C))	→	**NIL**
(MATCH '($X B $X) '(A B A))	→	**(T ((X . A)))**
(MATCH '(&- B C) '(A B C))	→	**(T NIL)**
(MATCH '(&-) '(A B C))	→	**(T NIL)**
(MATCH '(&X %(1 2 3) &Y) '(A B C 1 A B C D))	→	**(T ((Y A B C D) (X A B C)))**
(MATCH '(%(A (A $X) (A $X $-)) $X) '((A B C) B))	→	**(T ((X . B)))**
(MATCH '(&X 1 &X) '(A B C 1 A B C D))	→	**NIL**
(MATCH '(&X &X &X) '(A B C A B C A B C))	→	**(T ((X A B C)))**
(MATCH '($A &- ($- &B (&C) &D)) '(1 2 3 (4 5 (6 7))))	→	**(T ((D) (C 6 7) (B . 5) (A . 1)))**

Prenons comme exemple d'un petit programme utilisant cet algorithme de filtrage une fonction testant si un mot donné est un palindrome ou non. Rappelons qu'un palindrome est un mot qui, lu à l'envers, livre le même mot. Par exemple, le mot "otto" est un palindrome, de même le mot "aha", ou "rotor".[2]

Voici le début du programme :

[2] Georges Perec a écrit une petite histoire de plusieurs pages sous forme de palindrome, cf. *Oulipo, la littérature potentielle*, pp. 101-106, idées, Gallimard, 1973.

(DF PALINDROME (MOT) (PAL (EXPLODE (CAR MOT))))

Evidemment, ce programme ne fait qu'éclater le mot dans une suite de caractères pour que la fonction auxiliaire **PAL** puisse ensuite comparer le premier et le dernier caractère, et s'ils sont identiques comparer l'avant dernier caractère avec le deuxième, et ainsi de suite, jusqu'à ce qu'on rencontre deux caractères différents ou un seul caractère, ou bien plus rien.

Pour la comparaison du premier et du dernier caractère le filtre :

($X &MILIEU $X)

semble parfait : en plus, ce filtre récupère la suite de caractères qui reste si on enlève le caractère du début et de la fin :

```
(DE PAL (LISTE-DE-CARACTERES)
      (IF (NULL (CDR LISTE-DE-CARACTERES)) 'OUI
          (LET ((AUX (MATCH '($X &MILIEU $X) LISTE-DE-CARACTERES)))
               (IF (CAR AUX) ; que faire ??? ;
                   'NON))))
```

Le problème est : "que faire si le filtrage a réussi ?". A cet instant nous avons dans la variable **AUX** une liste de deux éléments : le premier élément est l'indicateur de réussite du filtrage et le deuxième élément est la A-liste résultant du filtrage. C'est cette A-liste qui nous indique, dans le couple **(MILIEU .** *valeur***)**, la suite de caractères qui doit encore être testée. Il suffirait alors de appeler récursivement **PAL** avec la valeur de **MILIEU**.

Nous pouvons le faire avec l'appel :

(PAL (CDR (ASSQ 'MILIEU (CADR AUX))))

ou, de manière plus élégante, en mettant l'appel récursif **(PAL MILIEU)** dans un environnement où les liaisons de la A-liste sont temporairement valides.

La fonction suivante construit l'environnement adéquat :

```
(DM LETALL (CALL)
   (LET ((ALIST (EVAL (CADR CALL))))
      (RPLACB CALL
           '((LAMBDA ,(MAPCAR ALIST 'CAR) ,@(CDDR CALL))
             ,@(MAPCAR ALIST (LAMBDA (X) (LIST QUOTE (CDR X))))))))
```

Cette macro crée, à partir d'une A-liste et d'une suite de commandes, une expression ou la suite de commandes est évaluée avec les liaisons déterminées par la A-liste.

Prenons un exemple : si la variable **L** contient la liste :

((X A B C) (Y . 1))

l'appel :

(LETALL L (PRINT X Y) (CONS Y X))

donne d'abord l'impression **(A B C) 1**, et livre ensuite la liste **(1 A B C)** et nous permet de compléter notre fonction **PAL** en insérant après **(IF (CAR AUX)** la commande :

(LETALL (CADR AUX) (PAL MILIEU))

Le filtrage, constituant de base de la plupart de langages de programmation issus de LISP, ouvre la voie à une programmation par règles. De tels systèmes interprètent un ensemble de couples

<situation, actions>. en les comparant à une situation donnée. L'interprète évalue la partie action du premier couple dont la description de situation est comparable à celle donnée.

Pour rapidement exposer cette technique, reconstruisons un petit programme Eliza, un des premiers programmes d'Intelligence Artificielle. Ce programme simule un psychiatre Rogerien. Le programme est constitué d'un ensemble de couples <filtre, action>. Après chaque énoncé de l'utilisateur (le patient), il compare la suite des filtres avec l'enoncé, et dès qu'un des filtres correspond, il imprime la partie action en y reprenant des parties de l'enoncé du patient. Voici un ensemble de telles couples :

filtre	*action*
(au revoir)	()
(stop)	()
(non)	'(vous êtes bien négatif)
(&- lisp &-)	'(alors vous savez programmer)
(&- ans &-)	'(remontez plus loin dans vos souvenirs)
(&- annees &-)	'(cela fait bien longtemps)
($-)	'(c est un peu court)
(&- je ne suis pas &l)	'(que vous fait pensez que vous n'êtes pas &l)
(&- suis &l)	'(pourquoi pensez-vous que vous êtes &l)
(&- je ne &-)	'(vous en êtes suˆr)
(&- non &-)	'(vous êtes très négatif)
(&- vous &-)	'(concentrez vous sur votre problème, svp)
(&- moi &-)	'(racontez-moi plus sur votre famille)
(&- je &-)	'(comment cela se manifeste-t-il)
(&- j ai &-)	'(depuis combien de temps)
(&- combien &-)	'(ce n est point la quantité qui compte)
($- $- $-)	'(hmmm)
(&-)	'(je ne vous suis pas très bien)

Graˆce à ce petit ensemble de règles, nous pouvons déjà faire des dialogues comme celui-ci (les impressions du programme sont en caractères **gras**, celle de l'utilisateur en caractères normaux) :

(depuis trois années je vis avec Marie)
(cela fait bien longtemps)
(pas du tout)
(hmmm)
(elle ne m'aime plus)
(vous en êtes suˆr)
(oui)
(c est un peu court)
(je ne suis pas court)
(que vous fait pensez que vous n'êtes pas court)
(vous vous moquez de moi)
(concentrez-vous sur votre problème, svp)
etc . . .

Bien que ce début de dialogue ait déjà l'air bien intéressant, il est nécessaire, afin de le rendre vraiment stimulant, d'acroître le nombre de couples <filtre, action> et d'améliorer le programme qui les utilise, dont voici un (à vous de l'implémenter, de l'améliorer et de le tester) :

```
(DE ELIZA ()
     (PRINT "bonjour, quel est votre problème ?")
     (WHILE (REPOND (ANALYSE (READ))))
     "au revoir")

(DE ANALYSE (PROBLEME)
    (LET ((PAT-REP LIST-PATTERN-REPONSE) (MATCH))
       (IF (CAR (SETQ MATCH (MATCH (CAAR PAT-REP) PROBLEME)))
            (REFLECHIT (CADR MATCH) (CDAR PAT-REP))
            (SELF (CDR PAT-REP) ()))))

(DE REFLECHIT (ENV MODELE)
      (EVAL '(LETALL ,ENV ,MODELE)))

(DE REPOND (REPONSE)
  (AND REPONSE (PRINT "PSY : " REPONSE))
```

La seule chose qui manque encore est de mettre la liste des filtres et réponses dans la variable globale **LIST-PATTERN-REPONSE** sous la forme :

```
(SETQ LIST-PATTERN-REPONSE
      '(  ((AU REVOIR) . ())
          ((STOP) . ())
          ((NON) . '(VOUS ETES BIEN NEGATIF))
          ((&- LISP &-) . '(ALORS VOUS SAVEZ PROGRAMMER))

          ; et ainsi de suite tous les couples donnés précédemment ;
```

Mais revenons à notre filtrage. Si nous pouvions accrocher des restrictions sur nos variables de filtre, l'écriture d'un programme comme **PAL** pourrait se simplifier encore énormément. Reprenons le palindrome : une autre manière de définir ce qu'est un palindrome est de dire que c'est une suite de lettres suivie de la même suite de lettres à l'envers, éventuellement séparée par une unique lettre. Ce qui pourrait être exprimer par :

(&X &X-*à-l'envers***)** ou **(&X $Y &X-***à-l'envers***)**

ou, de manière plus propre à LISP :

```
(OR (&X &(Y (EQUAL X (REVERSE Y)))) (&X $M &(Y (EQUAL X (REVERSE Y)))))
```

ou, de manière plus compacte :

```
(&X &(Z (LE (LENGTH Z) 1)) &(Y (EQUAL X (REVERSE Y))))
```

Introduire cette modification supplémentaire n'est pas difficile : il suffit d'ajouter un module qui traduit les variables des contraintes en leur valeur et évalue la contrainte à l'aide de la macro-fonction **LETALL**. Il est donc nécessaire d'introduire ce test de validité des contraintes dans la clause traitant les variables "élément" et celle des variables "segment".

```
((EQUAL (CAAR FILTRE) "$") (COND
      ((NULL DONNEE) ())
      ((EQ (CDAR FILTRE) '-)
            (MATCH-AUX (CDR FILTRE) (CDR DONNEE)))
      ; regardons si la variable a une liaison ;
      ((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
            (MATCH-AUX (CONS (CDR AUX) (CDR FILTRE)) DONNEE))
      ; il n'y a pas de liaison ;
      (T (SETQ ALIST (CONS (CONS (VAR (CDAR FILTRE)) (CAR DONNEE)) ALISTE))
            (IF (OR (NULL (CDDAR FILTRE))
                    (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))))
                (MATCH-AUX (CDR FILTRE) (CDR DONNEE))
                (NEXTL ALIST) NIL))))
```

La fonction **VAR** est nécessaire pour trouver la partie variable dans le filtre, étant donné que celle-ci peut être maintenant en position **CDR** (s'il n'y a pas des contraintes) ou **CADR** (s'il y a des contraintes) :

```
(DE VAR (FILTRE)
      (IF (ATOM (CDR FILTRE)) (CDR FILTRE) (CADR FILTRE)))
```

Et la nouvelle clause pour la variable segment devient :

```
((EQUAL (CAAR FILTRE) "&") (COND
   ; existe-t-il une valeur ;
   ((SETQ AUX (ASSQ (CDAR FILTRE) ALISTE))
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (CDR AUX) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))
      ; succès ;
      (IF (EQ (CDAR FILTRE) '-) ()
          (SETQ ALISTE (CONS (CONS (VAR (CDAR FILTRE)) DONNEE) ALISTE)))
          (IF (OR (NULL (CDDAR FILTRE))
                  (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))) T
              (NEXTL ALIST) NIL))
   (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()
          (SETQ ALISTE (CONS (LIST (VAR (CDAR FILTRE))) ALISTE)))
      (LET ((ALIST2 ALISTE))
           (IF DONNEE
               ; testons les contraintes ;
               (PROGN (IF (OR (NULL (CDDAR FILTRE))
                              (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))))
                          ; faisons les comparaisons successives ;
                          (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE)))
                      (IF AUX1 T
                          (SETQ ALISTE ALIST2)
                          (IF AUX (NEXTL DONNEE)
                              ; la liaison de la variable s'agrandit ;
                              (NCONC (CAR ALISTE) (LIST (NEXTL DONNEE))))
                          (SELF ALISTE)))
               ; échec total : restaurons la A-liste ;
               (IF AUX (NEXTL ALIST))
               NIL)))))
```

Avec cette petite modification nous pouvons comparer maintenant des structures avec des

contraintes sur les variables du filtre. Un ensemble de tels nouveaux filtres est présenté ci-dessous :

filtre	*donnée*	*résultat*
($X A B $(Y (EQ Y (+ 5 X))))	**(5 A B 10)**	**(T ((Y . 10) (X . 5)))**
($X A B $(Y (EQ Y (+ 5 X))))	**(5 A B 5)**	**NIL**
(&X A B &(Y (EQUAL Y (REVERSE Y))))	**(1 2 A B 2 1)**	**(T ((Y 2 1) (X 1 2)))**
(&X &(Y (EQUAL Y (REVERSE Y))))	**(A B C C B A)**	**(T ((Y C B A) (X A B C)))**

Un seul problème subsiste dans notre fonction de filtrage. Regardez l'exemple suivant :

(MATCH '(($X + $Y) * ($X - $Y)) '((A + B + C) * (A + B - C)))

D'après ce que nous savons, le résultat du filtrage doit être :

(T ((X A + B) (Y C)))

Néanmoins, notre fonction de filtrage ramène un échec. Pourquoi ? Regardez : le problème dans cette fonction vient de ce que nous utilisons la récursivité et le mécanisme de retour, généralement appelé *backtracking*. Ce backtracking, nous le faisons de manière explicite dans la clause concernant les variables de segment. Là, nous pouvons revenir vers l'arrière (et donc défaire des liasons sur la A-liste) jusqu'à ce que nous trouvions quelque chose qui marche, ou jusqu'à ce que nous ayons épuisé toutes les possibilités.

Malheureusement, dans la dernière clause nous interdisons les retours vers l'arrière : une fois terminée la comparaison du premier élément du filtre, ici **($X + $Y)**, avec le premier élément de la donnée, ici **(A + B + C)**, nous ne pouvons plus défaire la liaison résultante, même quand nous constatons par la suite, dans la comparaison des **CDR**s réspectifs, qu'elle est incorrecte.

Manquant de structures de contrôle plus sophistiquées[3], la seule solution qui nous reste, consiste à traduire les filtres et les données à l'entrée de la fonction **MATCH** en listes linéaires puis de retraduire les résultats à la sortie.

Ainsi, le filtre :

(($X + $Y) * ($X - $Y))

peut être linéarisé de la manière suivante :

(<< $X + $Y >> * << $X - $Y >>)

la donnée **((A + B + C) * (A + B - C))** peut être linéarisée en :

(<< A + B + C >> * << A + B - C >>)

et de cette façon, la dernière clause de notre fonction de filtrage, et donc la clause qui inhibe le retour en arrière, peut être éliminée, puisqu'elle ne sera plus nécessaire : tous les filtres et toutes les données seront linéaires !

Le programme de filtrage complet intégrant toutes nos modifications a maintenant la forme suivante :

(DE MATCH (FILTRE DONNEE)

[3] Nous examinerons de telles structures de contrôle dans un deuxième livre sur LISP.

```
  (LET ((ALISTE NIL))
    (IF (MATCH-AUX (LINEAR FILTRE) (LINEAR DONNEE))
        (LIST T ALIST) NIL)))

(DE MATCH-AUX (FILTRE DONNEE)
  (LET ((AUX NIL) (ALIST2 NIL))
  (COND
    ((NULL FILTRE)                  ; test de fin de recursion
     (NULL DONNEE))
    ((AND DONNEE (ATOM DONNEE))     ; donnée atomique
     ())                            ; échec
    ((ATOM (CAR FILTRE))            ; constante
     (AND (EQ (CAR FILTRE) (CAR DONNEE))     ; si égalité
          (MATCH-AUX (CDR FILTRE) (CDR DONNEE))))  ; on continue, sinon échec
((EQUAL (CAAR FILTRE) "%")          ; choix multiple
      (LET ((AUX (CDAR FILTRE)) (ALIST2 ALIST)) (COND
            ((NULL AUX) (SETQ ALIST ALIST2) NIL)
            ((MATCH-AUX (CONS (NEXTL AUX) (CDR FILTRE)) DONNEE) T)
            (T (SELF (CDR AUX) ALIST2)))))
((EQUAL (CAAR FILTRE) "$") (COND    ; variable élément
       ((NULL DONNEE) ())           ; échec
       ((EQ (CDAR FILTRE) '-) (NEXTL-DONNEE)  ; pas de aliste
            (MATCH-AUX (CDR FILTRE) DONNEE))
       ; regardons si la variable a une liaison ;
       ((SETQ AUX (ASSQ (VAR (CAR FILTRE)) ALIST))  ; le doublet
            (MATCH-AUX
              (IF (ATOM (CDR AUX))  ; à cause de la linéarisation
                  (CONS (CDR AUX) (CDR FILTRE))
                  (APPEND (LINEAR (LIST (CDR AUX))) (CDR FILTRE)))
              DONNEE))
       ; il n'y a pas de liaison, faut créer une nouvelle liaison ;
       (T (SETQ ALIST (CONS (CONS (VAR (CAR FILTRE)) (NEXTL-DONNEE)) ALIST))
          (IF   ; y a-t-il des contraintes
            (AND (CONSP (CDAR FILTRE)) (CDDAR FILTRE))
                 ; alors faut les évaluer
            (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))
                (MATCH-AUX (CDR FILTRE) DONNEE)  ; et continuer
                (NEXTL ALIST) NIL)       ; ça marche pas
            (MATCH-AUX (CDR FILTRE) DONNEE)))))  ; pas de contraintes
((EQUAL (CAAR FILTRE) "&") (COND    ; une variable segment
   ; existe-t-il une valeur ;
   ((SETQ AUX (ASSQ (VAR (CAR FILTRE)) ALIST))  ; le couple de la alist
      ; oui, comparer alors la valeur ;
      (MATCH-AUX (APPEND (LINEAR (CDR AUX)) (CDR FILTRE)) DONNEE))
   ; c'est la fin du filtre ? ;
   ((NULL (CDR FILTRE))       ; c'est la fin de la donnée
      ; alors succès ;
      (IF (EQ (CDAR FILTRE) '-) ()  ; rien à faire si anonyme
          ; sinon faut créer une liaison
          (SETQ ALIST (CONS
                        (CONS (VAR (CAR FILTRE)) (DELINEAR-DONNEE))
                        ALIST)))
          (IF          ; y a-t-il des contraintes ?
```

```
            (AND (CONSP (CDAR FILTRE))(CDDAR FILTRE))
            ; si oui faut les évaluer
            (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE))) T
                (NEXTL ALIST) NIL) ; échec : contrainte non satisfait
            T))  ; pas de contraintes : tout va bien
    (T (IF (SETQ AUX (EQ (CDAR FILTRE) '-)) ()  ; variable anonyme
           (SETQ ALIST (CONS (LIST (VAR (CAR FILTRE))) ALIST)))
       (LET ((ALIST2 ALIST)) ; bouclons jusqu'au match du reste
            (IF DONNEE    ; y a encore des données
              ; testons les contraintes ;
              (PROGN
                (IF (AND (CONSP (CDAR FILTRE))(CDDAR FILTRE))
                  (IF (EVAL '(LETALL ,ALIST ,@(CDDAR FILTRE)))
                    ; faisons les comparaisons successives ;
                    (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE))
                    NIL)  ; échec : contrainte non satisfaite
                  ; pas de contrainte : continuons
                  (SETQ AUX1 (MATCH-AUX (CDR FILTRE) DONNEE)))
                (IF AUX1 T  ; variable anonyme
                  (SETQ ALIST ALIST2) ; restaurons la alist
                  (IF AUX (NEXTL-DONNEE) ; avançons
                    ; la liaison de la variable s'agrandit ;
                    (NCONC (CAR ALIST) (LIST (NEXTL-DONNEE))))
                  (SELF ALIST)))  ; et continuons
              ; échec total : restaurons la A-liste ;
              (IF AUX (NEXTL ALIST))
              NIL))))))))

(DE VAR (FILTRE)       ; pour trouver la variable du filtre
  (IF (ATOM (CDR FILTRE)) (CDR FILTRE) (CADR FILTRE)))

(DE LINEAR (DONNEE)       ; pour linéariser
  (COND
    ((ATOM DONNEE)
     DONNEE)
    ((ATOM (CAR DONNEE))
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))
    ((MEMBER (CAAR DONNEE) '("$" "&" "%"))
     (CONS (CAR DONNEE) (LINEAR (CDR DONNEE))))
    (T
     (APPEND (CONS '<< (LINEAR (CAR DONNEE)))
          (CONS '>> (LINEAR (CDR DONNEE)))))))

(DE DELINEAR (DONNEE)       ; pour délinéariser
  (IF (ATOM DONNEE) DONNEE
    (DELINEAR-DONNEE)))

(DE DELINEAR-DONNEE ()       ; auxiliaire pour DELINEAR
  (COND
    ((ATOM DONNEE)
     DONNEE)
    ((EQ (CAR DONNEE) '<<)
     (NEXTL DONNEE)
```

```
     (CONS (DELINEAR-DONNEE) (DELINEAR-DONNEE)))
    ((EQ (CAR DONNEE) '>>)
     (NEXTL DONNEE)
     ())
    (T
     (CONS (NEXTL DONNEE) (DELINEAR-DONNEE)))))

(DE NEXTL-DONNEE ()       ; pour avancer dans les données linéarisées
  (IFN (EQ (CAR DONNEE) '<<) (NEXTL DONNEE)
    (NEXTL DONNEE)
    (DELINEAR-DONNEE)))


; les macro-caractères

(DMC "%" () (CONS '"%" (READ))) ; %X  -->  (% . X)

(DMC "$" () (CONS '"$" (READ))) ; $X  -->  ($ . X)

(DMC "&" () (CONS '"&" (READ))) ; &X  -->  (& . X)

; et finalement encore une fois la fameuse fonction LETALL

(DM LETALL (CALL)
  (LET ((ALIST (CADR CALL)))
    (RPLACB CALL
      '((LAMBDA ,(MAPCAR ALIST 'CAR) ,@(CDDR CALL))
           ,@(MAPCAR ALIST (LAMBDA (X) (LIST QUOTE (CDR X))))))))
```

Pour conclure ce livre, construisons un autre petit interprète de règles utilisant cet algorithme de filtrage. Ce programme utilise un ensemble de règles de la forme :

(<*filtre*> <*action*>**)**

où la partie *filtre* détermine *quand* il faut lancer une activité, et la partie *action* détermine *que faire* avec la donnée qui correspond au *filtre*.

Si nous appliquons notre interprète à la taˆche de simplifier des expressions algébriques, un exemple de filtre peut être :

($X + 0)

et la partie action correspondant peut alors être **X**, c'est évidemment une règle disant que **0** est élément neutre à droite pour l'addition.

L'activité de l'interprète se réduit à appliquer l'ensemble de règles jusqu'à ce que le résultat de cette application soit identique à l'expression originale, i.e.: jusqu'à ce que plus aucune règle puissse s'appliquer.

Voici la boucle top-level de notre interprète :

```
(DE INTERPRETE (EXPR EXP1 X)(COND
        ((EQUAL EXP1 EXPR) EXPR)
        (T (SETQ X (SIMPL EXPR REGLES))
            (INTERPRETE X EXPR))))
```

La première clause de cette fonction est la clause d'arrêt : l'interprète cesse d'appliquer l'ensemble des règles de réécritures quand l'expression calculée par la fonction auxiliaire **SIMPL** est égale à l'expression donnée à cette fonction.

La variable globale **REGLES** contient l'ensemble des règles. Si nous prenons l'exemple d'un simplificateur algébrique simple (i.e.: ne fonctionnant que sur des expressions algébriques complètement parenthésées), l'ensemble des règles peut alors être la liste suivante :

```
(SETQ REGLES '(
        (($(X (NUMBERP X)) + $(Y (NUMBERP Y))) (+ X Y))
        (($(X (NUMBERP X)) * $(Y (NUMBERP Y))) (* X Y))
        (($(X (NUMBERP X)) - $(Y (NUMBERP Y))) (- X Y))
        (($(X (NUMBERP X)) / $(Y (NUMBERP Y))) (/ X Y))
        (($X + 0)  X)
        (($X - 0)  X)
        ((0 + $X)  X)
        ((0 - $X)  (LIST '- X))
        (($X * 1)  X)
        ((0 * $X)  0)
        (($X * 0)  0)
        ((1 * $X)  X)
        (($X / 1)  X)
        (($X + $X) (LIST 2 '* X))
        (($X / $X) 1)
))
```

Mais il nous manque encore la fonction **SIMPL** qui doit effectivement balayer la liste des règles et, si une s'applique, livrer la réécriture correspondant et relancer l'ensemble des règles sur cette réécriture :

```
(DE SIMPL (EXPR REGL ALIST)(COND
    ((NULL EXPR)())
    ((ATOM EXPR) EXPR)
    ((CAR (SETQ ALIST (MATCH (CAAR REGL) EXPR)))
        (SIMPL (EVAL '(LETALL ,(CADR ALIST) ,(CADAR REGL))) REGLES))
    ((AND REGL (SIMPL EXPR (CDR REGL))))
    (T (CONS (SIMPL (CAR EXPR)  REGLES)
        (SIMPL (CDR EXPR)  REGLES)))))
```

C'est tout !

Voici une petite interaction avec ce mini-simplificateur :

```
? (INTERPRETE '(4 * (5 + 2)))
= 28
? (INTERPRETE '(4 * (A * 1)))
= (4 * A)
? (INTERPRETE '((A + 0) * ((B / B) * 1))))
= A
? (INTERPRETE '((A * (3 * 3)) - ((B + B) / (B - 0)))))
```

= ((A * 9) - ((2 * B) / B))

Si nous ajoutons à notre ensemble de règles les deux règles suivantes :

(($X * $Y) / $Y) X)
(($Y * $X) / $Y) X)

le dernier test nous livre alors :

? (INTERPRETE '((A * (3 * 3)) - ((B + B) / (B - 0)))))
= ((A * 9) - 2)

Ainsi, l'extension de cet interprète se fait simplement par adjonction de nouvelles règles. Bien suˆr, notre programme d'interprétation de règles n'est pas très efficace, néanmoins il est très général et peut s'appliquer à tout algorithme qui s'exprime par un ensemble de règles de réécriture : il suffit de changer l'ensemble des règles. Les améliorations possibles consistent à

1. adjoindre une possibilité de grouper des règles afin de ne pas balayer l'ensemble de toutes les règles à chaque itération,
2. adjoindre une possibilité de modifier l'ordre des règles dépendant des données pour accélerer la recherche d'une règle qui s'applique.
3. et, finalement, adjoindre une possibilité permettant pour chaque règle plusieurs filtres ainsi que plusieurs actions.

Notre livre d'introduction à la programmation en LISP s'achève ici. Nous n'avons vu qu'une minime partie de ce qui est possible et faisable dans la programmation en général, et dans le langage LISP en particulier. Seules les bases absolument nécessaires ont été exposées. Maintenant il s'agit de pratiquer ce langage : écrire ses propres programmes, lire et modifier des programmes écrits par d'autres.

Ce livre a remplit son rôle s'il vous a donné envie de programmer : à l'instar de toute technique, ce n'est pas la lecture d'un livre, aussi bon soit-il, qui permette l'apprentissage de la programmation : seule l'activité de votre programmation personnelle, l'observation de vos erreurs ainsi que leurs corrections et votre interaction avec l'ordinateur, vous permettront de devenir un programmeur expert.

chapitre précédent

chapitre suivant