14. LE FILTRAGE (PREMIERE PARTIE)

Dans la programmation en LISP il est très souvent nécessaire de chercher des éléments à l'intérieur d'une liste ou de comparer deux listes. Les fonctions que nous connaissons actuellement pour traiter ce type de problème, sont **EQUAL**, qui compare deux listes élément par élément, et les fonctions **MEMQ** et **MEMBER** qui cherchent respectivement l'occurrence d'un atome ou d'un élément quelconque à l'intérieur d'une liste donnée.

Bien que fort utiles, ces fonctions sont relativement restreintes et - surtout - n'aident pas à comparer deux listes approximativement : par exemple, savoir si une liste comporte, dans un certain ordre, quelques éléments particuliers, ou, savoir si une liste est composée de seulement trois éléments avec comme dernier élément un atome particulier.

Bien évidemment, nous pouvons toujours écrire un programme particulier pour résoudre chaque problème de comparaison indépendamment. Par exemple, pour savoir si une liste **L** comporte les trois éléments **DO**, **RE** et **MI** dans cet ordre, nous pouvons définir la fonction **DOREMI** que voici :

```
(DE DOREMI (L)
    (LET ((X (MEMQ 'DO L))) (COND
          ((NULL L) ())
          ((AND (EQ (CADR X) 'RE) (EQ (CADDR X) 'MI)) T))))
```

ou, pour savoir si une liste n'a que trois éléments avec comme dernier élément l'atome **MI**, nous pouvons construire la fonction **BIDON** que voici :

```
(DE BIDON (L)
  (IF (AND (= (LENGTH L) 3) (EQ (CAR (REVERSE L)) 'MI)) T NIL))
```

mais ce type de fonctions risque de se multiplier très vite, avec une fonction particulière pour chaque cas particulier qui se présente.

Le *filtrage* est une technique qui permet de comparer une liste à un "*canevas*" de liste, où le *canevas* est une manière de décrire la *structure* d'une liste. Pour reprendre les deux exemples ci-dessus, les canevas correspondants pourraient être

(& DO RE MI &)

et

($ $ MI)

Dans ces canevas, le signe '**&**' indique une suite d'éléments quelconque et le signe '**$**' indique un unique élément quelconque.

Ainsi, les canevas sont des descriptions, visuellement compréhensibles, de structures de listes. En termes plus techniques un tel canevas s'appelle un *filtre* ou un *pattern* et le processus de comparaison d'un *filtre* avec une liste s'appelle le *filtrage* ou le *pattern-matching*. Dans ce chapitre nous allons voir comment se construire un tel programme de *filtrage*.

14.1. PREMIERE VERSION D'UNE FONCTION DE FILTRAGE

Commençons par nous restreindre à des filtres composés exclusivement de *constantes* (c'est-à-dire des atomes quelconques) et des signes '**$**'. Voici quelques exemples de ce que notre fonction de comparaison **MATCH** doit pouvoir faire :

(MATCH '(DO RE MI) '(DO RE MI))	→	**T**
(MATCH '(DO RE MI) '(DO RE DO))	→	**NIL**
(MATCH '(DO $ MI) '(DO RE MI))	→	**T**
(MATCH '(DO $ MI) '(DO DO MI))	→	**T**
(MATCH '(DO $ MI) '(DO (DO RE MI) MI))	→	**T**
(MATCH '(DO $ MI) '(DO RE))	→	**NIL**
(MATCH '(DO $ MI) '(DO FA RE))	→	**NIL**
(MATCH '($ $ $) '(1 2 3))	→	**T**

Le premier filtre, **(DO RE MI)**, ne compare que des listes identiques puisqu'aucun des signes particuliers **$** ou **&** n'est utilisé.

Le deuxième filtre, **(DO $ MI)**, laisse passer toutes les listes à trois éléments qui débutent par l'atome **DO** et se terminent par l'atome **MI**. Le deuxième élément peut être n'importe quel atome ou n'importe quelle liste. Le signe **$** indique juste : qu'à l'endroit où il se trouve il doit y avoir *un* élément.

Le troisième, **($ $ $)**, filtre toute liste de trois éléments exactement.

Le filtrage, réduit au seul signe particulier '**$**', se comporte donc presque comme la fonction **EQUAL** : si aucune occurrence de '**$**' ne se trouve dans le filtre, le filtrage *est identique* au test d'égalité. S'il y a occurrence du signe '**$**' la comparaison peut juste 'sauter' l'élément correspondant dans la liste donnée. Avant d'écrire notre première version de la fonction **MATCH**, rappelons-nous la définition de la fonction **EQUAL** (cf §6.3) :

```
(DE EQUAL (ARG1 ARG2) (COND
      ((ATOM ARG1) (EQ ARG1 ARG2))
      ((ATOM ARG2) NIL)
      ((EQUAL (CAR ARG1) (CAR ARG2)) (EQUAL (CDR ARG1) (CDR ARG2)))))
```

Maintenant, l'écriture de **MATCH** ne pose plus de problèmes :

```
(DE MATCH (FILTRE DONNEE) (COND
      ((ATOM FILTRE) (EQ FILTRE DONNEE))
      ((ATOM DONNEE) NIL)
      ((EQ (CAR FILTRE) '$)
            (MATCH (CDR FILTRE) (CDR DONNEE)))
      ((MATCH (CAR FILTRE) (CAR DONNEE))
            (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

Cette fonction permet de filtrer avec le signe '**$**', donc d'extraire des données ayant le même nombre d'éléments que le filtre.

Si nous voulons également permettre le signe '**&**', qui remplace toute une *séquence* d'éléments, nous devons introduire une clause supplémentaire :

```
((EQ (CAR FILTRE) '&) . . .)
```

Afin de voir par quoi remplacer les '. . .', regardons d'abord un exemple. Le filtre :

(& DO &)

récupère toute liste qui contient au moins une occurrence de l'atome **DO**. Donc, ce filtre marche pour la liste **(DO)**, ainsi que pour toutes les listes où cet atome est précédé d'un nombre quelconque d'éléments, comme, par exemple, les listes

(1 DO)
(1 2 DO)
(1 1 2 DO)
(A B C D E F G DO)
etc

ainsi que pour les listes où cet atome est suivi d'un nombre quelconque d'éléments quelconques, comme, par exemple :

(DO 1)
(DO 1 2 3)
. . .
(1 DO 1)
(1 2 DO 1 2 3)
etc

Le signe '**&**' peut donc ne correspondre à rien (c'est le cas si la liste donnée est **(DO)**) ou à un nombre quelconque d'éléments.

La manière la plus simple d'implémenter cette forme de filtrage est d'utiliser récursivement la fonction **MATCH** elle-même. Ainsi, si nous rencontrons le signe '**&**' dans le filtre, nous allons d'abord essayer de filtrer le reste du filtre avec la donnée entière (en supposant alors qu'au signe '**&**' correspond la séquence vide) et, si ça ne marche pas, nous allons essayer de filtrer le reste du filtre avec les **CDR**s successifs de la donnée - jusqu'à ce que nous ayons trouvé une donnée qui fonctionne. Voici alors le nouveau code pour notre première version de **MATCH** :

```
(DE MATCH (FILTRE DONNEE) (COND
        ((ATOM FILTRE) (EQ FILTRE DONNEE))
        ((ATOM DONNEE) NIL)
        ((EQ (CAR FILTRE) '$)
            (MATCH (CDR FILTRE) (CDR DONNEE)))
        ((EQ (CAR FILTRE) '&) (COND
            ((MATCH (CDR FILTRE) DONNEE))
            (DONNEE (MATCH FILTRE (CDR DONNEE)))))
        ((MATCH (CAR FILTRE) (CAR DONNEE))
            (MATCH (CDR FILTRE) (CDR DONNEE)))))
```

Voici quelques exemples d'appels ainsi que leurs résultats :

(MATCH '(A B C) '(A B C)) → **T**
(MATCH '(A $ C) '(A B C)) → **T**
(MATCH '(A $ C) '(A 1 C)) → **T**
(MATCH '(A $ C) '(A 1 2 3 C)) → **NIL**
(MATCH '(A & C) '(A 1 2 3 C)) → **T**
(MATCH '(A & C) '(A C)) → **T**

Regardez bien la différence dans l'effet des deux signes '**$**' et '**&**' : l'un filtre *un unique* élément et l'autre filtre *une séquence* (éventuellement vide) d'éléments.

Faire des tentatives d'appels d'une fonction est très courante dans la programmation en intelligence

artificielle. Regardons donc une trace de l'exécution de l'appel

(MATCH '(A & C) '(A 1 2 3 C))

```
---> (MATCH (A & C) (A 1 2 3 C))      ; l'appel initial
  ---> (MATCH A A)                    ; le premier appel récursif
  <--- MATCH T                        ; ça marche
  ---> (MATCH (& C) (1 2 3 C))        ; le deuxième appel récursif
    ---> (MATCH (C) (1 2 3 C))        ; essayons la séquence vide
      ---> (MATCH C 1)                ; est-ce que ça marche ?
      <--- MATCH NIL                  ; non
    <--- MATCH NIL                    ; c'est pas la séquence vide
    ---> (MATCH (& C) (2 3 C))        ; essayons la séquence d'un élément
      ---> (MATCH (C) (2 3 C))        ; et répétons l'essai
        ---> (MATCH C 2)              ; ça s'annonce mal
        <--- MATCH NIL                ; évidemment !
      <--- MATCH NIL                  ; c'est pas la séquence à 1 élément
      ---> (MATCH (& C) (3 C))        ; essayons à 2 éléments
        ---> (MATCH (C) (3 C))
          ---> (MATCH C 3)
          <--- MATCH NIL              ; ça ne marche pas
        <--- MATCH NIL                ; non plus
        ---> (MATCH (& C) (C))        ; et avec 3 éléments ?
          ---> (MATCH (C) (C))        ; ça a l'air bien
            ---> (MATCH C C)          ; ici ça marche
            <--- MATCH T              ; réellement
            ---> (MATCH NIL NIL)      ; on est à la fin
          <------ MATCH T             ; et ça marchait réellement
 <---------------- MATCH T            ; enfin !
= T                                   ; voici le résultat final
```

Nous reviendrons au filtrage pour le généraliser et pour le rendre plus puissant. Pour cela, nous avons absolument besoin de revoir notre compréhension des listes. C'est cela que nous allons faire au chapitre suivant.

14.2. EXERCICES

1. Lesquels des appels récursifs de **MATCH** sont récursifs terminaux et lesquels ne le sont pas ?
2. Comment modifier le programme **MATCH** ci-dessus pour permettre l'utilisation de filtres donnant le choix entre plusieurs éléments possibles ? Par exemple, si nous voulons trouver à l'intérieur d'un programme tous les appels des fonctions **CAR** et **CDR** nous pouvons désigner ces appels par les deux filtres :

   (CAR &)
   (CDR &)

   Pour donner un seul filtre, par exemple :

   ((% CAR CDR) &)

   c'est-à-dire : chaque fois que notre fonction rencontre, dans le filtre, une sous-liste commençant par le signe '**%**', le reste de la liste est un ensemble d'éléments dont *un* doit se trouver à l'endroit correspondant dans la donnée.

   Voici quelques exemples de cette nouvelle fonction **MATCH** avec ce nouveau type de filtre :

(MATCH '(A (% B 1) C) *quelquechose*)

ne filtre que les deux listes **(A B C)** et **(A 1 C)**, et le filtre

(MATCH '((% A B) (% A B)) *quelquechose*)

ne filtre que les quatre listes : **(A A) (A B) (B A) (B B)**.

chapitre précédent

solutions

chapitre suivant