9. LES MEMO-FONCTIONS

Le chapitre 7 s'est terminé par un exercice d'écriture de la fonction **FIBONACCI**. Suivant la définition qui y était donnée, la forme la plus simple de cette fonction est :

```
(DE FIBONACCI (N)
   (COND
      ((= N 0) 1)
      ((= N 1) 1)
      (T (+ (FIBONACCI (1- N))
            (FIBONACCI (- N 2)) )) ))
```

Cette fonction est un très bel exemple d'une fonction récursive, mais malheureusement, elle est extrêmement inefficace. Examinons la : afin de calculer la valeur de **FIBONACCI** d'un nombre *n*, elle calcule la valeur de **FIBONACCI** de *n-1*, ensuite de *n-2* etc, jusqu'à ce qu'elle s'arrête avec *n* égal à 1. Les mêmes valeurs sont ensuite recalculées pour le deuxième appel récursif (à l'exception de **FIBONACCI** de *n-1*). Voici une trace d'un appel de cette fonction :

```
? (FIBONACCI 4)
  ---> FIBONACCI : (4)
    ---> FIBONACCI : (3)
      ---> FIBONACCI : (2)
        ---> FIBONACCI : (1)
        <--- FIBONACCI = 1
        ---> FIBONACCI : (0)
        <--- FIBONACCI = 1
      <--- FIBONACCI = 2
      ---> FIBONACCI : (1)
      <--- FIBONACCI = 1
    <--- FIBONACCI = 3      ; fin du premier appel récursif ;
    ---> FIBONACCI : (2)
      ---> FIBONACCI : (1)
      <--- FIBONACCI = 1
      ---> FIBONACCI : (0)
      <--- FIBONACCI = 1
    <--- FIBONACCI = 2      ; fin du deuxième appel récursif ;
  <--- FIBONACCI = 5
= 5
```

Intuitivement, il semble ahurissant que la machine recalcule des valeurs qu'elle a déjà, préalablement, calculées. Les *mémo-fonctions* font une utilisation intensive des P-listes pour y garder les valeurs déjà calculées des appels, donnant ainsi une possibilité de *souvenir* ou *mémorisation* à la fonction. Regardez, voici la définition de la fonction **FIBONACCI** comme mémo-fonction :

```
(DE FIB (N)
    (COND
        ((= N 0) 1)
        ((= N 1) 1)
        ((GET 'FIB N))      ; c'est ICI qu'on économise ! ;
        (T (PUT 'FIB N
                (+ (FIB (1- N))(FIB (- N 2))))
           (GET 'FIB N))))
```

Chaque fois, avant de calculer récursivement la valeur d'un appel, cette fonction regarde d'abord sur sa P-liste s'il n'y aurait pas déjà une valeur pour cet appel, si oui, **FIB** ramène cette valeur. Sinon, **FIB** calcule la valeur, la met sur la P-liste (pour des utilisations ultérieures) et la ramène également. Comparez la trace du calcul de **FIB** de 4 avec la trace obtenue pour la fonction **FIBONACCI** :

```
?  (FIB 4)
  ---> FIB : (4)
    ---> FIB : (3)
      ---> FIB : (2)
        ---> FIB : (1)
        <--- FIB = 1
        ---> FIB : (0)
        <--- FIB = 1
      <--- FIB = 2
      ---> FIB : (1)
      <--- FIB = 1
    <--- FIB = 3            ; fin du premier appel récursif ;
    ---> FIB : (2)
    <--- FIB = 2            ; fin du deuxième appel récursif ;
  <--- FIB = 5
=  5
```

Regardez maintenant la trace de l'appel de **FIB** de 5 suivant :

```
?  (FIB 5)
  ---> FIB : (5)
    ---> FIB : (4)      ; !! premier appel récursif ;
    <--- FIB = 5
    ---> FIB : (3)      ; !! deuxième appel récursif ;
    <--- FIB = 3
  <--- FIB = 8
=  8
```

Dès que la fonction rencontre l'appel récursif de **(FIB 4)**, elle peut immédiatement ramener la valeur 5, grâce au souvenirs mémorisés sur la P-liste de **FIB**. Voici sa P-liste après ces appels :

(5 8 4 5 3 3 2 2)

où vous trouvez sous l'indicateur 5 la valeur 8, sous l'indicateur 4 la valeur 5, sous l'indicateur 3 la valeur 3 et sous l'indicateur 2 la valeur 2. Ainsi, pour chaque appel de fonction qui a été déjà calculé une fois, le résultat est immédiatement trouvé en consultant la mémoire de la P-liste, et aucun calcul n'est effectué plusieurs fois.

Pensez à cette technique, si vous avez une fonction *très* récursive que vous utilisez beaucoup à l'intérieur de votre programme.

Il est encore possible d'améliorer cette fonction **FIB**, en commençant par mettre sur la P-liste de **FIB** les valeurs pour 0 et 1, ce qui permettra d'enlever les deux tests :

((= N 0) 1)
((= N 1) 1)

et accélérera le calcul.

Ci-dessous la fonction **FIB** ainsi modifiée :

```
(DE FIB (N)
  (IF (<= N 1) 1     ; pour éviter des problèmes ! ;
      (UNLESS (GET 'FIB 0)
              (PUT 'FIB 0 1)
              (PUT 'FIB 1 1))
      (LET ((N N)) (COND
             ((GET 'FIB N))
             (T (PUT 'FIB N
                     (+ (SELF (1- N)) (SELF (- N 2))))
                (GET 'FIB N))))))
```

Dans cette version nous avons utilisé une nouvelle fonction : **UNLESS**. De même que la fonction **IF**, c'est une fonction de séléction. Sa définition est :

(UNLESS *test action*$_1$ *action*$_2$ . . . *action*$_n$**)**

→ évalue *action*$_1$ *à* *action*$_n$ et ramène *action*$_n$ en valeur si *test* = **NIL**

→ **NIL** si *test* ≠ **NIL**

l'appel

(UNLESS *test action*$_1$ *action*$_2$ . . . *action*$_n$**)**

est donc identique à l'appel de la fonction **IF** suivant :

(IF *test* **()** action$_1$ action$_2$ . . . action$_n$**)**

et n'est qu'une simplification de celui-ci.

Nous avons également une fonction **WHEN** qui est l'inverse de **UNLESS** :

(WHEN *test action*$_1$ *action*$_2$ . . . *action*$_n$**)**

→ évalue *action*$_1$ *à* *action*$_n$ et ramène *action*$_n$ en valeur si *test* ≠ **NIL**

→ **NIL** si *test* = **NIL**

L'appel

(WHEN *test action*$_1$ *action*$_2$ . . . *action*$_n$**)**

peux donc être écrit comme :

(UNLESS (NULL *test***)** $action_1$ $action_2$ . . . $action_n$**)**

ou encore comme :

(IF (NULL *test***) ()** $action_1$ $action_2$ . . . $action_n$**)**

Pour écrire vos sélections, vous avez donc le choix entre les fonctions **IF**, **COND**, **UNLESS** et **WHEN**.

9.1. EXERCICES

1. Ecrivez la fonction **FACTORIELLE** comme mémo-fonction. Comparez les temps d'exécution de **FACTORIELLE** avec utilisation de la P-liste avec ceux de la fonction sans utilisation de la P-liste.
2. Regardez le programme suivant :

```
(DE FOO (X) (COND
     ((< X 0) (- 0 X))
     (T (COND ((ZEROP X) 0)
          (T (* X -1))))))
```

a. Simplifiez ce programme, en ne prenant en considération que la structure syntaxique de ce programme, de manière à n'utilisez qu'un seul **COND**.

b. Connaissant le sens des différentes fonctions arithmétiques, simplifiez le programme de manière telle qu'il n'y ait plus de **COND** du tout.

3. Ecrivez un petit simplificateur algébrique. Ce simplificateur doit savoir simplifier des expressions arithmétiques LISP (donc des expressions algébriques bien parenthésées) et connaître - au minimum - les simplifications suivantes :

(+ e 0)	→	e
(+ 0 e)	→	e
(* e 1)	→	e
(* 1 e)	→	e
(* e 0)	→	0
(* 0 e)	→	0
(+ *nombre1 nombre2*)	→	*nombre1* + *nombre2*
(* *nombre1 nombre2*)	→	*nombre1* * *nombre2*

Ainsi, le programme devra, par exemple, simplifier l'expression

(+ (* x 0) (* 10 (+ y 0)))

en

(* 10 y)

chapitre pécédent

solutions

chapitre suivant