

11. LES DIVERS TYPES DE FONCTIONS UTILISATEURS : DE et DF

11.1. LA FONCTION DE

Pour définir une fonction utilisateur nous utilisons la fonction LISP standard **DE**. Telle que nous l'avons vue, **DE** définit une fonction d'un certain **nom** et d'un certain nombre de *paramètres* : autant de paramètres qu'il y a de variables dans la liste de variables.

A l'appel d'une telle fonction, chaque argument est évalué et lié à la variable correspondante. Par exemple, si nous définissons la fonction **FOO** comme :

```
(DE FOO (X Y Z) (CADR (PRINT (LIST X Y Z))))
```

nous décrivons une fonction de nom **FOO**, à trois arguments qui s'appellent respectivement **X**, **Y** et **Z**, et qui imprime les valeurs des arguments. C'est à dire, si nous appelons cette fonction avec :

```
(FOO (CAR '(A B C)) 1 2)
```

d'abord, la variable **X** est liée à l'atome **A** (la valeur du premier argument), la variable **Y** est liée au nombre **1** (la valeur du deuxième argument), et la troisième variable est liée à la valeur du troisième argument, le nombre **2**. Ensuite, le corps de la fonction est évalué avec les variables liées à ces valeurs. L'effet est donc l'impression de la liste :

```
(A 1 2)
```

et le résultat de cet appel est la valeur numérique **1**.

La fonction **DE** permet donc de définir des fonctions utilisateurs à nombre fixe d'arguments, et dont chacun des arguments est évalué à l'appel. Bien souvent, il peut nous arriver de vouloir définir une fonction avec un nombre arbitraire d'arguments, comme la fonction standard **PRINT** qui permet n arguments (avec un $n \in [1, \infty[$). Supposons alors que la fonction standard **+** n'admet que deux arguments et que nous désirons disposer d'une fonction **PPLUS** qui additionne un nombre quelconque d'entiers. Une première solution à ce problème serait de définir la fonction ci-dessous :

```
(DE PPLUS (L)  
  (IF (NULL L) 0  
      (+ (CAR L) (PPLUS (CDR L)))))
```

Mais cette fonction demande une liste d'arguments à l'appel. Ainsi il faut l'appeler par :

```
(PPLUS '(1 2 3 4 5 6))
```

ou

```
(PPLUS '(-1 2 -3 4 -5 6 -7 8))
```

En réalité, il serait préférable d'écrire les arguments l'un après l'autre, comme ci-dessous :

(PLUS 1 2 3 4 5 6)

ou

(PLUS -1 2 -3 4 -5 6 -7 8)

En LISP, il suffit d'écrire :

(DE PLUS L (PPLUS L))

la fonction **PPLUS** étant définie ci-dessus.

Regardez bien la différence de cette définition de fonction avec toutes les autres que nous connaissons : au lieu d'une *liste de variables* nous avons juste un *atome variable*.

Si vous définissez une fonction dont le nom-de-fonction est suivi d'un atome, cet atome est, à l'appel de la fonction, lié à la *liste de toutes les valeurs des différents arguments*.

Dans la fonction **PLUS** ci-dessus, la seule et unique variable, **L**, est, lors de l'appel de :

(PLUS 1 2 3 4 5 6)

liée à la liste des valeurs des différents arguments, donc à la liste :

(1 2 3 4 5 6)

Cette liste est ensuite envoyée à la fonction **PPLUS** qui, quant à elle, attend une liste des nombres pour en calculer leur somme.

Prenons un instant un autre exemple : vous avez sûrement constaté qu'il est bien difficile de construire une liste nouvelle à partir d'une suite d'éléments. Chaque fois que nous étions dans une telle situation, nous avons dû écrire quelque chose comme

(CONS élément₁ (CONS élément₂ ... (CONS élément_n ()) ...))

Par exemple, afin de construire à partir des atomes **A**, **B** et **C** la liste (**A B C**), nous sommes amenés à écrire :

(CONS 'A (CONS 'B (CONS 'C NIL)))

Connaissant maintenant la possibilité de définir des fonctions à un nombre quelconque d'arguments, nous pouvons très simplement écrire une fonction **LIST** qui construit une liste constituée des valeurs de ses différents arguments. La voici :

(DE LIST L L)

C'est tellement simple que cela paraît incroyable ! Regardons ce qui se passe dans l'appel :

(LIST 'A 'B 'C)

1. D'abord la variable **L** est liée à la liste des valeurs des arguments. Les arguments sont '**A**', '**B**' et '**C**' respectivement. Les valeurs de ces arguments sont donc respectivement **A**, **B** et **C**, et la variable **L** est donc liée à la liste (**A B C**).
2. Le corps de cette fonction est réduit à l'expression **L**, donc à une évaluation de la variable **L**. La valeur de cette évaluation est la valeur de l'appel de la fonction **LIST**. Bien entendu, la valeur de l'évaluation d'une variable est la valeur à laquelle elle est liée. Cet appel ramène donc en valeur la liste (**A B C**), la liste que nous voulions construire !

Vous voyez l'utilité de cette chose ?

Notez quand même que nous avons un problème avec cette forme de définition de fonction : nous ne savons pas comment faire des appels récursifs. Afin de voir pourquoi, reprenons la petite fonction **PLUS** ci-dessus et essayons de ne pas utiliser la fonction auxiliaire **PPLUS** :

```
(DE PLUS L
  (IF (NULL L) 0
    (+ (CAR L) (PLUS (CDR L)))))
```

Cette écriture, bien qu'ayant l'air toute naturelle, est évidemment fausse !

Si vous ne savez pas pourquoi, ne continuez pas tout de suite à lire, essayez plutôt de trouver l'erreur vous-mêmes !

Regardez, à l'appel initial :

```
(PLUS 1 2 3 4)
```

la variable **L** est liée à la *liste* **(1 2 3)**, par le simple mécanisme mis en marche par cette sorte de définition de fonction. Etant donné que cette liste n'est pas vide, LISP procède donc à l'évaluation de la ligne :

```
(+ (CAR L) (PLUS (CDR L)))
```

qui devrait donc calculer la somme de **1** et du résultat de l'appel récursif de **PLUS** avec l'argument **(2 3 4)**, le **CDR** de la valeur de la variable **L**. MAIS !!! Quelle horreur ! Ce n'est plus une suite de nombre qui est donnée à la fonction **PLUS** mais une liste. Nous savons que **L** est liée à la liste des valeurs des arguments : **L** recevra donc une liste contenant un seul et unique élément qui est la valeur de l'argument **(2 3 4)**, ce qui n'est sûrement pas ce qu'on voulait faire (Quelle est la valeur de **(2 3 4)** ? sûrement pas la suite des nombres 2, 3 et 4 !).

C'est cette difficulté qui nous a incité, dans l'écriture de la fonction **PLUS**, à déléguer la boucle (la répétition) vers une fonction auxiliaire, en l'occurrence la fonction **PPLUS**.

Rappelez-vous bien de ce problème d'impossibilité de faire des appels récursifs simples chaque fois que vous construisez une fonction à nombre d'arguments quelconque. Bien entendu, dans la suite de ce livre vous trouverez une solution au problème des appels récursifs de ce type de fonction.

En LISP les fonctions utilisateurs définies avec la fonction **DE** s'appellent des *EXPRs*, et, dans des anciens dialectes de LISP, tels que MACLISP par exemple, les fonctions à nombre d'arguments quelconque s'appellent des *LEXPRs* ou des *NEXPRs*.

11.2. LA FONCTION DF

De même qu'il est intéressant de disposer parfois d'une fonction à nombre quelconque d'arguments, où chacun des arguments est évalué, on a de temps à autre besoin de fonctions à nombre quelconque d'arguments *sans* évaluation des arguments. De telles fonctions peuvent être réalisées grâce à la fonction de définition de fonctions utilisateurs **DF**, qui est définie comme suit :

```
(DF nom-de-fonction (variable) corps-de-la-fonction)
```

Syntaxiquement, **DF** est identique à **DE**, avec, en VLISP, la seule exception qu'une fonction définie par **DF** n'a qu'une seule et unique variable paramètre. A l'appel, cette variable est liée à la liste des arguments *non évalués*, donc : la liste des arguments tels quels.

En `LE_LISP`, `DF` peut avoir plusieurs variables. La liaison se fait *exactement* de la même manière que pour `DE`, sauf que les divers arguments ne sont pas évalués.

Voici l'exemple le plus simple :

```
(DF QLIST (L) L)      en VLISP
(DF QLIST L L)       en LE_LISP
```

Cette fonction ressemble fortement à la fonction `LIST` ci-dessus, puisque, comme dans `LIST`, le corps de la fonction se réduit à l'évaluation de l'unique variable paramètre. Etant donné que les arguments ne seront *pas* évalués, l'appel

```
(QLIST A B C D)
```

ramène donc la liste des arguments tels quels, c'est-à-dire :

```
(A B C D)
```

Notez que les divers arguments *n'ont pas* été quotés. Si on l'appelle comme :

```
(QLIST 'A 'B 'C)
```

le résultat est la liste :

```
('A 'B 'C)
```

Nous verrons dans la suite l'intérêt de ce type de fonctions et comment on peut sélectivement évaluer l'un ou l'autre des arguments.

Les fonctions utilisateurs définies avec la fonction `DF` s'appellent des *FEXPRs*.

11.3. EXERCICES

1. Définissez une fonction `TIMES` à nombre d'arguments quelconque qui calcule le produit de tous ses arguments. Exemples :

```
(TIMES 1 2 3 4)          → 24
(TIMES (CAR '(1 2 3)) (CADR '(1 2 3)) (CADDR '(1 2 3))) → 6
```

2. Définissez une fonction `SLIST`, à nombre d'arguments quelconque, qui construit une liste contenant *les valeurs* des arguments de rang pair. Exemples :

```
(SLIST 1 2 3 4 5 6)     → (2 4 6)
(SLIST 1 2 3 4 5 6 7)  → (2 4 6)
(SLIST 1 'A 2 'B 3 'C)  → (A B C)
```

3. Définissez également une fonction `QLIST`, à nombre d'arguments quelconque, qui construit une liste contenant les arguments de rang pair (tels quels). Exemples :

```
(QLIST A B C D E F)     → (B D F)
(QLIST A B C D E F G)   → (B D F)
```

4. Finalement, définissez une fonction `MAX` qui trouve le maximum d'une suite de nombres et une fonction `MIN` qui trouve le minimum d'une suite de nombres. Voici quelques appels exemples :

```
(MAX 1 2 10 -56 20 6)   → 20
(MIN 1 2 10 -56 20 6)   → -56
(MAX (MIN 0 1 -34 23 6) (MIN 36 37 -38 476 63)) → -34
```