7. L'ARITHMETIQUE

7.1. LES FONCTIONS ARITHMETIQUES DE BASE

Bien que LISP soit principalement un langage de programmation symbolique (non numérique) il donne des possibilités de programmer des problèmes numériques. Ici nous nous limiterons au calcul avec des nombres entiers compris entre -2^{15} et $2^{15} - 1$.[1] Regardons d'abord les fonctions-arithmétiques standard :

(1+ *n***)** $\rightarrow$ $n + 1$

La fonction **1+** additionne donc la valeur **1** à son argument. C'est une fonction d'*incrémentation*.

(1- *n***)** $\rightarrow$ $n - 1$

La fonction **1-** soustrait la valeur **1** à son argument. C'est une fonction de *décrémentation*.

Attention 1

Remarquons que **1+** et **1-** sont des *noms* de fonction qui s'écrivent comme toutes les fonctions LISP en préfixe, c.à.d.: avant l'argument. **1+** ou **1-** sont donc des noms comme **CAR** ou **CDR**, et le signe '+' ou le signe '-' doit être écrit de manière telle qu'il suive directement le chiffre '1' (sans espace entre '1' et '+' ou '-' !).

Attention 2

Notons également que ni la fonction **1+**, ni la fonction **1-**, ni les autres fonctions arithmétiques de base ne modifient les valeurs de leurs arguments : tout ce qu'elles font est de calculer une valeur !

Voici les fonctions d'addition (+), de soustraction (-), de multiplication ($*$) et de division (/) :[2]

(+ *n1 n2***)**	$\rightarrow$	$n1 + n2$
(- *n1 n2***)**	$\rightarrow$	$n1 - n2$
(* *n1 n2***)**	$\rightarrow$	$n1 * n2$
(/ *n1 n2***)**	$\rightarrow$	$\lfloor n1 / n2 \rfloor$

Toutes ces opérations arithmétiques sont génériques, c.à.d. les arguments peuvent être d'un type numérique quelconque : si toutes les arguments sont des nombres entiers, le résultat est également un nombre entier, sinon, le résultat s'adapte au type des arguments. L'exception à cette règle est la fonction de division / : En VLISP cette division est une *division entière*, ce qui veut dire que le résultat d'une division de *n1* par *n2* est

[1] La taille minimum et maximum des nombres dépend de la machine particulière sur laquelle votre LISP tourne. Ici nous comptons avec une machine à mots de 16 bits.

[2] Dans LE_LISP et dans quelques versions de VLISP (par exemple en VLISP-10), les fonctions +, - et * admettent un nombre quelconque d'arguments. Avant de commencer à écrire de grands programmes sur une machine, testez donc d'abord ces quelques fonctions arithmétiques.

le plus grand entier *x*, tel que

$$(x * n2) \leq n1$$

ce qui se note normalement : $\lfloor n1 / n2 \rfloor$.

En LE_LISP, par contre, le résultat d'une division, par la fonction /, d'un nombre entier *n* par un entier *m* produit une interruption si *n* n'est pas un multiple de **m**. Pour avoir la division entière, telle que nous venons de la décrire, dans le cas de deux arguments entier, LE_LISP offre la fonction **QUO**.

Dans ce livre d'introduction, nous nous limitons au calcul avec des entiers.

Il existe également une fonction pour calculer le *reste de la division entière*, nommé **REM**.

(REM *n1 n2***)** $\rightarrow$ $n1 - ((\lfloor n1 / n2 \rfloor) * n2)$

Voici quelques exemples numériques :

(1+ 1024)	$\rightarrow$	**1025**
(1- 0)	$\rightarrow$	**-1**
(1- -345)	$\rightarrow$	**-346**
(+ 17 9)	$\rightarrow$	**26**
(+ 1 -1)	$\rightarrow$	**0**
(- 10 24)	$\rightarrow$	**-14**
(* 6 4)	$\rightarrow$	**24**
(/ 1024 2)	$\rightarrow$	**512**
(/ 17 7)	$\rightarrow$	**2**
(REM 1024 2)	$\rightarrow$	**0**
(REM 17 7)	$\rightarrow$	**3**
(+ (* 3 3)(* 4 4))	$\rightarrow$	**25**
(- 111 (* 4 (/ 111 4)))	$\rightarrow$	**3**
(/ (+ 10 90)(- (* 5 6) 10))	$\rightarrow$	**5**

Munis de ces quelques fonctions standard nous pouvons écrire déjà un grand nombre de fonctions utiles. Voici par exemple une fonction qui calcule le *carré* de son argument :

```
(DE CARRE (N) (* N N))
```

et voici encore quelques fonctions arithmétiques simples et fort utiles : d'abord deux fonctions qui calculent le *carré* du *carré* de son argument :

```
(DE CARRE-DU-CARRE (N) (CARRE (CARRE N)))
```

ou, plus simplement :

```
(DE CARRE-DU-CARRE (N) (* N (* N (* N N))))
```

Voici une fonction qui calcule l'expression (A + B)/(A - B) :

```
(DE SOMME-DIFFER (N M)(/ (+ N M)(- N M)))
```

que fait donc la fonction suivante ?

```
(DE FOO (L1 L2)
    (+ (* (CAR L1)(CAR L2))(* (CADR L1)(CADR L2))))
```

7.2. LES PREDICATS NUMERIQUES

Afin de pouvoir écrire des fonctions arithmétiques plus intéressantes, nous devons préalablement connaitre quelques prédicats numériques. Voici les plus importants :

D'abord un prédicat qui teste si son argument est égal à 0

(ZEROP *n***)**
- → **0** si $n = 0$
- → **NIL** si $n \neq 0$

le prédicat **=** se lit *numériquement égal* et teste l'égalité de deux nombres

(= *arg1 arg2***)**
- → **T** si *arg1 = arg2*
- → **NIL** s'ils sont différents

le prédicat **>** se lit *greater* (en français : strictement supérieur)

(> *n1 n2***)**
- → *n1* si *n1* est supérieur à *n2*
- → **NIL** si *n1* est inférieur ou égal à *n2*

le prédicat **<** se lit *less* ou *smaller* (en français : strictement inférieur)

(< *n1 n2***)**
- → *n1* si *n1* est inférieur à *n2*
- → **NIL** si *n1* est supérieur ou égal à *n2*

le prédicat **>=** se dit *greater or equal*

(>= *n1 n2***)**
- → *n1* si *n1* est supérieur ou égal à *n2*
- → **NIL** si *n1* est inférieur à *n2*

et le prédicat **<=** se prononce *less or equal*[3]

(<= *n1 n2***)**
- → *n1* si *n1* est inférieur ou égal à *n2*
- → **NIL** si *n1* est supérieur à *n2*

Naturellement, le prédicat **EQ** fonctionne aussi bien sur des nombres entiers que sur des atomes

[3] Dans quelques versions de VLISP, la fonction >= s'appelle **GE** et la fonction <= s'appelle **LE**.

quelconques.

Dans la foulée, construisons deux autres prédicats, un, appelé **EVENP**, qui teste si son argument est un nombre pair, et un autre, appelé **ODDP**, qui teste si son argument est un nombre impair. Ils seraient donc définis comme suit :

(EVENP *n***)**
→ *n1* si *n* est pair
→ **NIL** si *n* est impair

(ODDP *n***)**
→ *n1* si *n* est impair
→ **NIL** si *n* est pair

L'écriture de ces deux fonctions ne devrait pas poser de problèmes :

```
(DE EVENP (N) (IF (ZEROP (REM N 2)) N NIL))

(DE ODDP (N) (IF (= (REM N 2) 1) N NIL))
```

7.3. LA PROGRAMMATION NUMERIQUE

Pour nous habituer un tant soit peu aux algorithmes numériques nous allons, dans le reste de ce chapitre, donner quelques-uns des algorithmes les plus connus. Commençons par l'omniprésente fonction **FACTORIELLE**. Pour ceux qui ne savent pas ce qu'est la factorielle d'un nombre *n*, rappelons que c'est le produit de tous les nombres entiers positifs inférieurs ou égaux à *n*. Ainsi on a les définitions suivantes :

factorielle (n) = n ∗ (n - 1) ∗ (n - 2) ∗ ... ∗ 3 ∗ 2 ∗ 1
factorielle (0) = 1

Dans les manuels de mathématiques, vous trouvez très souvent la définition récurrente que voici :

factorielle (0) = 1
factorielle (n) = n ∗ factorielle (n - 1)

Cette définition nous donne - tel quel - l'algorithme récurrent à construire. Traduisons-la en LISP sous forme récursive :

```
(DE FACTORIELLE (N)
    (IF (ZEROP N) 1
        (* N (FACTORIELLE (1- N)))))
```

C'est, à quelques transpositions textuelles près, la même chose que la définition mathématique. Juste pour le plaisir voici quelques appels et leurs résultats :

(FACTORIELLE 0) → **1**
(FACTORIELLE 1) → **1**
(FACTORIELLE 2) → **2**
(FACTORIELLE 3) → **6**
(FACTORIELLE 4) → **24**
(FACTORIELLE 5) → **120**

Cette fonction croit très vite.[4] Pensez-y quand vous la testez, sachant que si vous travaillez sur une machine à 16 bits, LISP ne connait pas les nombres entiers supérieurs à $2^{15} - 1$!

Dans son temps, Euclide avait developpé un algorithme pour trouver le *plus grand commun diviseur*, abrégé PGCD.[5] Son algorithme est le suivant :

pour calculer le PGCD de *m* et *n* il faut
si *m* est supérieur à *n* calculer le PGCD de *n* et *m*
sinon, si le reste de la division de *n* par *m* est égal à 0
si *m* = 1 alors *m* et *n* sont premiers entre eux
sinon *m* est le PGCD de *m* et *n*
sinon on a la relation :
PGCD (m, n) = PGCD (reste $\lfloor$(n / m)$\rfloor$, m)

Encore une fois : les définitions récurrentes semblent être très courantes en mathématiques. Voici la traduction de l'algorithme d'Euclide en fonction LISP récursive :

```
(DE PGCD (M N)
    (LET ((X (PGCD1 M N)))
        (IF X X (LIST M 'ET N 'SONT 'PREMIERS 'ENTRE 'EUX))))

(DE PGCD1 (M N)(COND
      ((> M N) (PGCD1 N M))
      ((= (REM N M) 0)
         (IF (= M 1) () M))
      (T (PGCD1 (REM N M) M)) ))
```

Evidemment, vous ne connaissez pas la fonction **LET**. **LET** est utilisée ici pour garder la valeur de l'appel **(PGCD1 M N)** quelque part, afin d'y avoir accès dans la suite. Il aurait été possible d'écrire la fonction **PGCD** comme suit :

---

[4] D'ailleurs, saviez vous qu'il n'existe qu'un seul nombre pour lequel l'équation :

factorielle(n) = factorielle(a) ∗ factorielle(b)

a une solution non trivial ? C'est

factorielle(10) = factorielle(7) * factorielle(6)

[5] Vous trouverez de nombreux renseignements sur l'algorithme d'Euclide et d'autres algorithmes numériques dans le deuxième volume de l'oeuvre monumentale de D. E. Knuth. Ce livre est une telle source d'informations que sa lecture est devenue quasiment obligatoire. Voici les références : D. E. Knuth, *The Art of Computer Programming*, le premier volume s'intitule *Fundamental Algorithms*, le deuxième *Semi-numerical Algorithms* et le troisième *Searching and Sorting*. Le tout est publié chez Addison-Wesley Publ. Company.

```
(DE PGCD (M N)
    (IF (PGCD1 M N) (PGCD1 M N)
        (LIST M 'ET N 'SONT 'PREMIERS 'ENTRE 'EUX)))
```

Mais avec cette écriture l'appel de **(PGCD1 M N)** aurait été calculé *deux* fois : une fois pour le prédicat du **IF**, pour savoir s'il existe un PGCD de **M** et **N**, et une fois pour réellement connaître la valeur de l'appel. Ce qui - le moins qu'on puisse dire - ne serait pas très élégant. On avait besoin d'une méthode pour *lier* cette valeur intérmédiaire à une variable. C'est justement l'une des utilisations courantes de la fonction **LET**.

Voici comment elle fonctionne : **LET** est une manière de définir une fonction *sans nom*. Voici sa définition syntaxique :

(LET (*élé*$_1$ *élé*$_2$ ... *élé*$_n$**)** *corps-de-fonction***)**

avec chacun des *élé*$_i$ défini comme :

(*variable valeur***)**

La sémantique - i.e.: le sens - de la fonction **LET** est la suivante :

> A l'entrée du **LET** les différentes variables sont liées aux valeurs données dans les couples *variables-valeurs* de la liste de variables du **LET**. Ensuite est evalué le *corps-de-fonction* du **LET** avec ces liaisons. La valeur d'un **LET** est la valeur de la dernière expression évaluée (comme pendant l'évaluation de fonctions utilisateurs normales). C'est donc à la fois la définition d'une fonction (sans nom toutefois) et son appel !

Revenons alors à la fonction **PGCD** : à l'entrée du **LET** la variable **X** sera liée à la valeur de l'appel de la fonction auxiliaire **PGCD1**. Cette valeur peut être un nombre, si les arguments ont un PGCD, ou **NIL** si les arguments sont premiers entre eux. Selon cette valeur de **X**, on ramènera alors en valeur soit le PGCD soit une liste indiquant que les deux arguments sont premiers entre eux. Voici quelques appels et leurs valeurs :

```
? (PGCD 256 1024)
= 256

? (PGCD 3456 1868)
= 4

? (PGCD 567 445)
= (567 ET 445 SONT PREMIERS ENTRE EUX)

? (PGCD 565 445)
= 5

? (PGCD 729 756)
= 27

? (PGCD 35460 18369)
= 9
```

La fonction **LET** est - nous l'avons dit - à la fois la définition et l'appel d'une fonction. Naturellement,

comme toute fonction, une fonction définie avec **LET** peut être appelée récursivement. Pour cela, nous avons besoin d'une fonction particulière puisqu'aucun nom pour nommer la fonction est disponible (rappelons que les fonctions construites avec **LET** n'ont *pas* de noms). En VLISP, la fonction **SELF** sert à appeler (sans préciser de nom) la fonction *à l'intérieur de laquelle elle se trouve*. Afin de voir comment ça marche, regardons l'exemple d'une fonction trouvant la racine entière d'un nombre. Rappelons, que la racine entière *x* d'un nombre *n* est définie comme le plus grand entier *x* tel que

$$(x * x) \leq n$$

(le signe '≤' veut dire *est inférieur ou égal à*). Voici la définition LISP de la fonction **RACINE-ENTIER**

```
(DE RACINE-ENTIER (N)
    (LET ((P 1)(K 0)(N (1- N)))
        (IF (< N 0) K
            (SELF (+ P 2)(+ K 1)(- N (+ 2 P))))))
```

La fonction **SELF** renvoie à la fonction construite à l'aide de **LET**. Tout se passe comme si la fonction **LET** définissait une fonction appelée **SELF** *suivie* immédiatement de son appel avec les arguments donnés dans le **LET**. On pourrait donc s'imaginer que **LET** (dans cet exemple précis) définit d'abord une fonction :

```
(DE  SELF (P K N)
     (IF (< N 0) K
         (SELF (+ P 2)(+ K 1)(- N (+ 2 P)))))
```

et modifie la fonction **RACINE -ENTIER** en ceci :

```
(DE RACINE-ENTIER (N)
     (SELF 1 0 (1- N)))
```

CE N'EST PAS CE QUI SE PASSE EN REALITE, mais c'est un bon modèle pour comprendre comment ça fonctionne !

Note important concernant LE_LISP :

Pour éviter quelques problèmes inhérants à la fonction **SELF**, LE_LISP a remplacé cette fonction par une deuxième construction du style **LET**. C'est la fonction **LETN**, pour **LET N**ommé. La définition syntaxique de cette fonction est comme suit :

(LETN *nom* **(***élé*$_1$ *élé*$_2$ ... *élé*$_n$**)** *corps-de-fonction***)**

Cette forme a le même comportement que la fonction **LET** *sauf* que le symbole *nom* est, à l'intérieur de **LETN** lié à cette expression même. On peut donc, récursivement appeler ce **LET** par un appel de la fonction *nom*.

En LE_LISP, vous utilisez donc deux formes de **LET** : la première, **LET**, si vous voulez juste temporairement lier des valeurs *sans* faire des appels récursifs, la deuxième, **LETN**, si vous voulez faire des appels récursifs de la fonction. En LE_LISP, la fonction **RACINE-ENTIER** s'écrit donc de la manière suivante :

```
(DE RACINE-ENTIER (N)
    (LETN SELF ((P 1)(K 0)(N (1- N)))
        (IF (< N 0) K
            (SELF (+ P 2)(+ K 1)(- N (+ 2 P))))))
```

Dans la suite de ce livre, si vous programmez en LE_LISP, il faut alors remplacer toutes les occurrences de

(LET . . . (SELF . . .) . . .)

par

(LETN SELF . . . (SELF . . .) . . .)

Remplacez donc toute expression **LET** qui contient un appel de la fonction **SELF** à l'intérieur, par une expression **LETN** avec le premier argument **SELF** en laissant le reste de l'expression tel quel.

Quel est l'algorithme sous-jacent dans **RACINE-ENTIER** ? Pourquoi cette fonction livre-t-elle la racine entière ? Pour montrer que ça marche réellement, voici quelques exemples :

```
(RACINE-ENTIER 9)       →    3
(RACINE-ENTIER 10)      →    3
(RACINE-ENTIER 8)       →    2
(RACINE-ENTIER 49)      →    7
(RACINE-ENTIER 50)      →    7
(RACINE-ENTIER 12059)   →  109
```

Voici encore quelques exemples de fonctions numériques. D'abord la fonction **DEC-BIN** qui traduit un nombre décimal en une liste de zéro et de un (donc en une liste représentant un nombre binaire) :

```
(DE DEC-BIN (N)
    (IF (> N 0)
        (APPEND (DEC-BIN (/ N 2))(APPEND (REM N 2) NIL))
         NIL))
```

Pour cette fonction, la définition de la fonction **APPEND** (cf. chapitre 6.2) a été légèrement modifiée Quelle est la nature de cette modification ? Pourquoi est-elle nécessaire ?

Voici quelques exemples d'appels :

```
? (DEC-BIN 5)
= (1 0 1)

? (DEC-BIN 1023)
= (1 1 1 1 1 1 1 1 1 1)

? (DEC-BIN 1024)
= (1 0 0 0 0 0 0 0 0 0 0)

? (DEC-BIN 555)
= (1 0 0 0 1 0 1 0 1 1)
```

Si vous n'avez pas encore trouvé la modification nécessaire de la fonction **APPEND**, pensez au fait qu'elle

doit concaténer deux *listes*, et qu'ici le premier argument peut être un atome, comme dans l'appel **(APPEND (REM N 2) NIL)**.

Voici donc la nouvelle fonction **APPEND** :

```
(DE APPEND (ELEMENT LISTE)(COND
    ((NULL ELEMENT) LISTE)
    ((ATOM ELEMENT) (CONS ELEMENT LISTE))
    (T (CONS (CAR ELEMENT) (APPEND (CDR ELEMENT) LISTE)))))
```

Naturellement, si nous avons une fonction de traduction de nombres décimaux en nombres binaires, il nous faut aussi la fonction inverse : la fonction **BIN-DEC** traduit des nombres binaires en nombres décimaux.

```
(DE BIN-DEC (N)
    (LET ((N N)(RES 0))
        (IF N (SELF (CDR N)
                    (+ (* RES 2)(CAR N)))
              RES)))))
```

quelques exemples d'utilisation :[6]

```
? (BIN-DEC '(1 0 1))
= 5

? (BIN-DEC '(1 0 0 0 0 0 0 0 0 0 0))
= 1024

? (BIN-DEC '(1 0 1 0 1 0 1 0 1 0 1))
= 1365

? (BIN-DEC '(1 1 1))
= 7
```

Etudiez ces fonctions attentivement ! Faites les tourner à la main, faites les tourner sur la machine.

7.4. EXERCICES

1. Ecrivez une fonction, nommée **CNTH**, à deux arguments : un nombre *n* et une liste *l*, et qui livre à l'appel le *n*-ième élément de la liste *l*.

   Voici quelques appels possibles de cette fonction :

---

[6] Rappelons que, d'après notre algorithme de traduction, le programme **BIN-DEC** s'écrit en LE_LISP comme suit:

```
(DE BIN-DEC (N)
    (LETN SELF ((N N)(RES 0))
        (IF N (SELF (CDR N)
                    (+ (* RES 2)(CAR N)))
              RES)))))
```

(CNTH 2 '(DO RE MI FA)) → **RE**
(CNTH 3 '(DO RE MI FA)) → **MI**
(CNTH 1 '((TIENS TIENS) AHA AHA)) → **(TIENS TIENS)**

2. Ecrivez une fonction de transformation de nombres décimaux en nombres octaux et hexadécimaux.
3. Ecrivez une fonction qui traduit des nombres octaux en nombres décimaux, et une autre pour traduire des nombres hexadécimaux en nombres décimaux.
4. En 1220, un grand problème mathématique dans la région de Pise était le suivant : si un couple de lapins a chaque mois deux enfants, un mâle et une femelle, et si après un mois les nouveaux couples peuvent également produire deux petits lapins de sexe opposé, combien de lapins existent alors après n mois ? Monsieur Leonardo de Pise (communément surnommé Fibonacci) trouva la règle récurrente suivante :

fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

Ecrivez une fonction qui calcule les nombres de Fibonacci pour tout argument *n* positif.

5. Que fait la fonction suivante :

```
(DE QUOI (N)
    (LET ((RES 0) (N N))
        (IF (> N 0)
            (SELF (+ (* RES 10)(REM N 10)) (/ N 10))
            RES)))
```

6. La fonction **ACKERMANN** est définie comme suit :

ack(0, n) = n + 1
ack(m, 0) = ack(m - 1, 1)
ack(m, n) = ack(m - 1, ack(m, n - 1))

Ecrivez cette fonction en LISP. Avez vous une idée de ce qu'elle fait ?

7. Ecrivez la fonction **LENGTH** qui calcule le nombre d'éléments d'une liste. Voici quelques exemples d'appel :

(LENGTH '(PEU A PEU)) → **3**
(LENGTH '(((ART) ADJ) NOM VERBE)) → **3**
(LENGTH '((DO RE MI))) → **1**
(LENGTH ()) → **0**

8. Ecrivez une fonction qui calcule le nombre d'atomes d'une liste. Exemples :

(NBATOM '(PEU A PEU)) → **3**
(NBATOM '(((ART) ADJ) NOM VERBE)) → **4**
(NBATOM '((DO RE MI))) → **3**
(NBATOM '()) → **0**

chapitre précédent | solutions | chapitre suivant