INRIA

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tel (1) 39 63 55 11

Rapports de Recherche

N° 1016

Programme 1

BigNum : UN MODULE PORTABLE ET EFFICACE POUR UNE ARITHMETIQUE A PRECISION ARBITRAIRE

Jean Claude HERVE
François MORAIN
David SALESIN
Bernard Paul SERPETTE
Jean VUILLEMIN
Paul ZIMMERMANN

Avril 1989

3065

BigNum: Un module portable et efficace pour une arithmétique à précision arbitraire

BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic

Jean Claude Hervé *
David Salesin *
Jean Vuillemin *

François Morain †
Bernard Paul Serpette †
Paul Zimmermann †

Resumé

Nous spécifions un module Le-Lisp d'arithmétique entière à précision arbitraire, portable, mais néanmoins efficace. La rapidité d'exécution de ce module est assujettie à la réécriture en langage machine du noyau du module; nous fournissons les codes assembleurs pour les machines VAX, Mips, 68020 et NS. Ce module est du domaine public pour toute utilisation non commerciale.

Abstract

We specify a Le-Lisp package for arbitrary-precision integer arithmetic that is portable, yet efficient. Making the package run fast on a given computer involves re-writing a small kernel of our package in native assembly language. We provide such assembly code for VAX, Mips, 68020 and NS instruction sets. The package is publicly available for non-commercial uses.

*Digital Equipment Corp., Paris Research Laboratory, 85 Av. Victor Hugo. 92500 Rueil-Malmaison, France.

†Institut National de Recherche en Informatique et Automatique, 78153, Rocquencourt, France.

Un rapport décrivant le même module pour le langage C est disponible, en anglais, en tant que rapport DEC. Cette implantation étant particulièrement dédiée au langage C, la nôtre dédiée à Le-Lisp[1], certaines différences de spécifications seront décrites. L'implantation C sera appelée BigNum.c.

1 Introduction

Le développement d'un module arithmétique à précision arbitraire, devant être à la fois efficace et portable, soulève deux problèmes principaux:

- Un module arithmétique, écrit dans un langage de haut niveau (C, Le-Lisp, Modula2+, ...) est typiquement quatre à dix fois plus lent que le même module écrit directement en langage machine.
- La plupart des opérations arithmétiques sont accélérées d'un facteur non négligeable lorsque les allocations mémoire sont enlevées des boucles principales, les résultats intermédiaires et finaux utilisant l'espace mémoire référencé par certains des paramètres des fonctions d'appel.

Pour résoudre ces problèmes, nous avons organisé notre logiciel en deux niveaux:

1. Un niveau appelé **Bn**, dans lequel chaque fonction traite des entiers non signés, sans allocation, et retournant les résultats en lieu et place des premiers arguments passés à la fonction.
2. Un niveau appelé **Bz**, implanté au dessus de **Bn**, spécifiant une arithmétique signée et où, d'une manière traditionnelle, est faite l'allocation des résultats.

Pour des raisons d'efficacité, le module **Bn** est lui-même structuré en deux niveaux:

- Le noyau **KerN**, contenant les primitives dont le temps d'exécution est critique.
- Le reste de **Bn**, dont le code Le-Lisp fait appel au noyau.

Le noyau **KerN** est écrit en C et en Le-Lisp pour des raisons de portabilité et de documentation et peut être compilé tel quel. Néanmoins, pour obtenir une implantation vraiment efficace sur une machine donnée, **KerN** doit être écrit directement en langage machine. Nous en proposons les versions pour les machines VAX, Mips, 68020, et NS. **KerN** reste de taille raisonnable: 550 lignes de C, 350 lignes de Le-Lisp, 700 lignes d'assembleur 68020 et 600 lignes d'assembleur VAX.
La partie de **Bn** ne faisant pas partie du noyau est écrite directement en Le-Lisp et en C. Une fonction est considérée comme faisant parti du noyau s'il est possible de gagner un gain d'efficacité de plus de 20%, sur un test standard, en récrivant cette fonction en assembleur. La connaissance exacte des procédures faisant partie de **KerN** n'a d'importance que pour les personnes devant porter le module sur une nouvelle machine, ou ceux qui ne sont pas satisfaits de l'implantation C ou Le-Lisp.
Finalement, notons que **Bn** sert de base à d'autres modules spécialisés tels qu'arithmétique rationnelle, polynômiale ou modulaire.

2 Représentation des nombres

Les objets de base traités par **KerN** sont les entiers naturels représentés par leurs décompositions dans une base nommée B; ainsi les nombres de **KerN** s'écrivent sous la forme:

$$N = \sum_{i=0}^{i<n} N_i B^i \ .$$

Nous faisons la restriction que la base B est une puissance de 2. Il existe un entier T tel que: $B = 2^T$ ou $T = \log_2(B)$, T détermine le nombre de bits nécessaires pour représenter un chiffre N_i. La valeur de l'entier

[1] Le-Lisp est une marque déposée de l'INRIA.

T est disponible par la constante **BN_DIGIT_SIZE**.

Par la suite nous utiliserons la notion de *sous-nombre* d'un nombre. Le sous-nombre, noté $N_{nd,nl}$, du nombre

$$N = \sum_{i=0}^{i<n} N_i B^i$$

est :

$$N_{nd,nl} = \sum_{i=0}^{i<nl} N_{nd+i} B^i \ .$$

Ceci impose que $nd \geq 0$, $nl \geq 1$ et $nd + nl \leq n$; n est appelé la taille du nombre N (par la suite nous noterons $n = size(N)$). Pour le cas particulier $nl = 0$ nous convenons que tous les sous-nombres $N_{i,0}$ sont équivalents à 0. On remarquera que le chiffre N_i est équivalent au sous-nombre $N_{i,1}$.
Toutes les primitives supposent, sans le tester, que, si $N_{i,j}$ est un sous-nombre du nombre N de taille n, alors les préconditions suivantes sont vérifiées: $0 \leq i < n$, $0 \leq j \leq n$ et $0 \leq i + j \leq n$. Aucune de ces préconditions ne sont explicitement testées. L'usage de ces routines, hors de ces conditions de validité, peut conduire à des violations de mémoire.
Dans l'implantation BigNum.c un sous-nombre $N_{nd,nl}$ est converti en un autre sous-nombre équivalent $M_{0,nl}$ moyennant l'affectation $M = N + nd$; ceci impose d'une part que le langage permet l'utilisation de pointeur à l'intérieur de tableau de chiffres[2], et d'autre part que l'organisation interne des chiffres dans un nombre soit fixée. Ces restrictions permettent de réduire (généralement d'un tiers) les paramètres formels des primitives. Ainsi tous les sous-nombres sont spécifiés par deux variables (N et nl) au lieu de trois (N, nd et nl).

3 Opération en place : Bn

Dans cette section nous décrirons les fonctions traitant des entiers non signés. Dans les exemples les entiers sont imprimés en hexadécimal, les poids forts à gauche, chaque chiffre étant encadré par le caractère |. Nous utiliserons une implantation où **BN_DIGIT_SIZE** vaut 32.

3.1 Addition

(BnAddCarry *N nd nl r***)**

La fonction **BnAddCarry** propage en place la retenue r, valant 0 ou 1, sur le sous-nombre $N_{nd,nl}$ et retourne la retenue sortante. Plus formellement la fonction **BnAddCarry** effectue l'opération suivante :

$$N_{nd,nl} + r = (\sum_{i=0}^{i<nl} N_{nd+i} B^i) + r = (\sum_{i=0}^{i<nl} N'_{nd+i} B^i) + r' B^{nl} \ .$$

La fonction **BnAddCarry** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et retourne comme résultat la retenue r' valant 0 ou 1. Cette retenue vaut 1 si et seulement si le sous-nombre $N_{nd,nl}$ est égal à $B^{nl} - 1$ et si la retenue entrante r vaut 1. La longueur nl peut-être égale à zéro, dans ce cas la retenue sortante prend la même valeur que la retenue entrante.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (BnComplement n 0 4) n)
= |FFFFFFFF|FFFFFFFF|FFFFFFFF|FFFFFFFF|
? (cons (BnAddCarry n 1 3 0) n)
```

[2] Addition d'un pointeur sur des chiffres et d'un index comme dans l'exemple $N + nd$.

```
= (0 . |FFFFFFFF|FFFFFFFF|FFFFFFFF|FFFFFFFF|)
? (cons (BnAddCarry n 2 2 1) n)
= (1 . |00000000|00000000|FFFFFFFF|FFFFFFFF|)
? (cons (BnAddCarry n 0 3 1) n)
= (0 . |00000000|00000001|00000000|00000000|)
? (cons (BnAddCarry n 0 3 1) n)
= (0 . |00000000|00000001|00000000|00000001|)
? (cons (BnAddCarry n 0 3 1) n)
= (0 . |00000000|00000001|00000000|00000002|)
```

(**BnAdd** *N nd nl M md ml r*)

La fonction **BnAdd** effectue l'addition des deux sous-nombres $N_{nd,nl}$ et $M_{md,ml}$ et de la retenue r, place le résultat dans le sous-nombre $N_{nd,ml}$, propage la retenue de cette addition sur le sous-nombre $N_{nd+ml,nl-ml}$ et retourne la retenue sortante. Plus formellement la fonction **BnAdd** effectue l'opération suivante:

$$N_{nd,nl} + M_{md,ml} + r = (\sum_{i=0}^{i<nl} N_{nd+i}B^i) + (\sum_{i=0}^{i<ml} M_{md+i}B^i) + r = (\sum_{i=0}^{i<nl} N'_{nd+i}B^i) + r'B^{nl} \ .$$

La fonction **BnAdd** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et retourne comme résultat la retenue r' valant 0 ou 1. Ceci impose donc que $0 \leq r \leq 1$ et $nl \geq ml$. Les deux nombres N et M peuvent être identiques [3] à la condition que $nd \leq md$. De même la longueur ml peut-être égale à zéro, dans ce cas la forme (**BnAdd** *N nd nl M md 0 r*) est équivalente à: (**BnAddCarry** *N nd nl r*).

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (BnComplement n 0 3) n)
= |00000000|FFFFFFFF|FFFFFFFF|FFFFFFFF|
? (cons (BnAdd n 0 1 n 1 1 1) n)
= (1 . |00000000|FFFFFFFF|FFFFFFFF|FFFFFFFF|)
? (cons (BnAdd n 0 2 n 2 1 0) n)
= (1 . |00000000|FFFFFFFF|00000000|FFFFFFFE|)
? (cons (BnAdd n 2 1 n 3 1 1) n)
= (1 . |00000000|00000000|00000000|FFFFFFFE|)
? (cons (BnAdd n 0 2 n 0 1 0) n)
= (0 . |00000000|00000000|00000001|FFFFFFFC|)
```

3.2 Soustraction

(**BnComplement** *N nd nl*)

La fonction **BnComplement** remplace tous les chiffres du sous-nombre $N_{nd,nl}$ par leur complément à la base, ce qui correspond à l'inversion logique de tous les bits. Plus formellement la fonction **BnComplement** effectue l'opération suivante:

$$\overline{N_{nd,nl}} = \overline{\sum_{i=0}^{i<nl} N_{nd+i}B^i} = \sum_{i=0}^{i<nl} \overline{N_{nd+i}}B^i = \sum_{i=0}^{i<nl} (B - N_{nd+i} - 1)B^i \ .$$

[3] Au sens de la fonction eq de Le-Lisp: (eq *N M*).

La fonction **BnComplement** ne retourne aucun résultat significatif. Aucun effet de bord n'est effectué lorsque la longueur *nl* vaut 0.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnComplement n 1 2) n)
= |00000003|FFFFFFFD|FFFFFFFE|00000000|
```

(**BnSubtractBorrow** *N nd nl r*)

La fonction **BnSubtractBorrow** propage en place l'emprunt r, de valeur 0 ou 1, sur le sous-nombre $N_{nd,nl}$ et retourne l'emprunt sortant. Plus formellement la fonction **BnSubtractBorrow** effectue l'opération suivante:

$$N_{nd,nl} + B^{nl} + r - 1 = (\sum_{i=0}^{i<nl} N_{nd+i}B^i) + B^{nl} + r - 1 = (\sum_{i=0}^{i<nl} N'_{nd+i}B^i) + r'B^{nl} .$$

La fonction **BnSubtractBorrow** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et retourne comme résultat l'emprunt sortant r' valant 0 ou 1. Cet emprunt vaut 0 si et seulement si le sous-nombre $N_{nd,nl}$ et l'emprunt entrant sont égaux à 0. La longueur *nl* peut-être égale à zéro, dans ce cas l'emprunt sortant prend la même valeur que l'emprunt entrant.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) n)
= |00000000|00000000|00000000|00000000|
? (cons (BnSubtractBorrow n 1 3 1) n)
= (1 . |00000000|00000000|00000000|00000000|)
? (cons (BnSubtractBorrow n 2 2 0) n)
= (0 . |FFFFFFFF|FFFFFFFF|00000000|00000000|)
? (cons (BnSubtractBorrow n 0 3 0) n)
= (1 . |FFFFFFFF|FFFFFFFE|FFFFFFFF|FFFFFFFF|)
? (cons (BnSubtractBorrow n 0 3 0) n)
= (1 . |FFFFFFFF|FFFFFFFE|FFFFFFFF|FFFFFFFE|)
? (cons (BnSubtractBorrow n 0 3 0) n)
= (1 . |FFFFFFFF|FFFFFFFE|FFFFFFFF|FFFFFFFD|)
```

(**BnSubtract** *N nd nl M md ml r*)

La fonction **BnSubtract** effectue la soustraction des deux sous-nombres $N_{nd,nl}$ et $M_{md,ml}$ et de l'emprunt r, place le résultat dans le sous-nombre $N_{nd,ml}$, propage l'emprunt de cette soustraction sur le sous-nombre $N_{nd+ml,nl-ml}$ et retourne l'emprunt sortant. Plus formellement la fonction **BnSubtract** effectue l'opération suivante:

$$N_{nd,nl} - M_{md,ml} + B^{ni} + r - 1 = (\sum_{i=0}^{i<nl} N_{nd+i}B^i) - (\sum_{i=0}^{i<ml} M_{md+i}B^i) + B^{nl} + r - 1 = (\sum_{i=0}^{i<nl} N'_{nd+i}B^i) + r'B^{nl} .$$

La fonction **BnSubtract** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et retourne comme résultat l'emprunt sortant r' valant 0 ou 1. Ceci impose donc que $0 \leq r \leq 1$ et $nl \geq ml$. Les deux nombres

N et M peuvent être identiques à la condition que $nd \leq md$. De même la longueur ml peut-être égale à zéro, dans ce cas la forme (**BnSubtract** *N nd nl M md 0 r*) est équivalente à: (**BnSubtractBorrow** *N nd nl r*).

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (BnComplement n 3 1) n)
= |FFFFFFFF|00000000|00000000|00000000|
? (cons (BnSubtract n 0 1 n 1 1 0) n)
= (0 . |FFFFFFFF|00000000|00000000|FFFFFFFF|)
? (cons (BnSubtract n 0 2 n 2 1 0) n)
= (1 . |FFFFFFFF|00000000|00000000|FFFFFFFE|)
? (cons (BnSubtract n 2 1 n 3 1 1) n)
= (0 . |FFFFFFFF|00000001|00000000|FFFFFFFE|)
? (cons (BnSubtract n 0 3 n 0 1 0) n)
= (1 . |FFFFFFFF|00000000|FFFFFFFF|FFFFFFFF|)
```

3.3 Multiplication

(**BnMultiplyDigit** *P pd pl N nd nl M md*)

La fonction **BnMultiplyDigit** effectue la multiplication du sous-nombre $N_{nd,nl}$ par le chiffre M_{md} avec accumulation dans le sous-nombre $P_{pd,pl}$, et retourne la retenue produite. Plus formellement la fonction **BnMultiplyDigit** effectue l'opération suivante:

$$P_{pd,pl} + M_{md} \times N_{nd,nl} = (\sum_{i=0}^{i<pl} P_{pd+i}B^i) + M_{md}(\sum_{i=0}^{i<nl} N_{nd+i}B^i) = (\sum_{i=0}^{i<pl} P'_{pd+i}B^i) + r'B^{pl} .$$

La fonction **BnMultiplyDigit** remplace tous les chiffres P_{pd+i} par leurs équivalents P'_{pd+i} et retourne comme résultat la retenue sortante r' valant 0 ou 1. Ceci impose donc que $pl > nl$. La longueur nl peut-être égale à zéro, dans ce cas la retenue sortante vaut zéro et aucun effet de bord n'est effectué. Les deux nombres P et N peuvent être identiques à la condition que $pd \leq nd$. Enfin le chiffre M_{md} peut-être n'importe lequel des chiffres de P ou de N. Les cas $M_{md} = 0$ et $M_{md} = 1$ sont explicitement testés pour accélérer les temps de calcul.

Exemple:

```
? (progn (setq n (BnCreate 'n 6)) (BnComplement n 0 4) n)
= |00000000|00000000|FFFFFFFF|FFFFFFFF|FFFFFFFF|FFFFFFFF|
? (cons (BnMultiplyDigit n 3 3 n 0 2 n 0) n)
= (0 . |FFFFFFFF|00000000|00000000|FFFFFFFF|FFFFFFFF|FFFFFFFF|)
? (cons (BnMultiplyDigit n 3 3 n 0 2 n 0) n)
= (1 . |FFFFFFFD|FFFFFFFF|00000001|FFFFFFFF|FFFFFFFF|FFFFFFFF|)
? (cons (BnMultiplyDigit n 3 3 n 0 2 n 0) n)
= (1 . |FFFFFFFC|FFFFFFFE|00000002|FFFFFFFF|FFFFFFFF|FFFFFFFF|)
? (cons (BnMultiplyDigit n 0 3 n 0 2 n 3) n)
= (1 . |FFFFFFFC|FFFFFFFE|00000002|00000001|FFFFFFFF|FFFFFFFD|)
? (cons (BnMultiplyDigit n 0 2 n 0 1 n 3) n)
= (1 . |FFFFFFFC|FFFFFFFE|00000002|00000001|00000001|FFFFFFF7|)
? (cons (BnMultiplyDigit n 0 2 n 0 1 n 3) n)
= (0 . |FFFFFFFC|FFFFFFFE|00000002|00000001|00000003|FFFFFFE5|)
```

(**BnMultiply** *P pd pl N nd nl M md ml*)

La fonction **BnMultiply** effectue la multiplication des deux sous-nombres $N_{nd,nl}$ et $M_{md,ml}$ avec accumulation dans le sous-nombre $P_{pd,pl}$, et retourne la retenue produite. Plus formellement la fonction **BnMultiply** effectue l'opération suivante:

$$P_{pd,pl} + N_{nd,nl} \times M_{md,ml} = (\sum_{i=0}^{i<pl} P_{pd+i}B^i) + (\sum_{i=0}^{i<nl} N_{nd+i}B^i)(\sum_{i=0}^{i<ml} M_{md+i}B^i) = (\sum_{i=0}^{i<pl} P'_{pd+i}B^i) + r'B^{pl} .$$

La fonction **BnMultiply** remplace tous les chiffres P_{pd+i} par leurs équivalents P'_{pd+i} et retourne comme résultat la retenue sortante r' valant 0 ou 1. Ceci impose donc que $pl \geq nl + ml$. La longueur nl peut-être égale à zéro, dans ce cas la retenue sortante vaut zéro et aucun effet de bord n'est effectué. La boucle de calcul étant effectué sur ml, la fonction est plus performante lorsque $nl \geq ml$. Il n'est pas possible d'effectuer des chevauchements entre les sous-nombres $P_{pd,pl}$ et $N_{nd,nl}$ ou $M_{md,ml}$.

Exemple:

```
? (progn (setq n  (BnCreate 'n 8)) (for (i 0 1 7) (BnSetDigit n i i)) n)
= |00000007|00000006|00000005|00000004|00000003|00000002|00000001|00000000|
? (cons (BnMultiply n 4 4 n 0 2 n 2 2) n)
= (0 . |00000007|00000009|00000007|00000004|00000003|00000002|00000001|00000000|)
? (progn (BnComplement n 0 4) n)
= |00000007|00000009|00000007|00000004|FFFFFFFC|FFFFFFFD|FFFFFFFE|FFFFFFFF|
? (cons (BnMultiply n 4 4 n 0 2 n 2 2) n)
= (1 . |00000003|00000008|0000000D|00000007|FFFFFFFC|FFFFFFFD|FFFFFFFE|FFFFFFFF|)
```

(**BnShiftLeft** *N nd nl M md s*)

La fonction **BnShiftLeft** décale vers la gauche tous les chiffres du sous-nombre $N_{nd,nl}$, les s bits laissés vacants d'un chiffre sont remplacés par les s bits sortants du chiffre précédent, les s bits laissés vacants du premier chiffre sont remplacés par des zéros, et les s bits sortants du dernier chiffre sont placés, cadrés sur les poids faibles, dans le chiffre M_{md}. La fonction **BnShiftLeft** effectue donc une multiplication par 2^s sur le sous-nombre $N_{nd,nl}$. Plus formellement la fonction **BnShiftLeft** effectue l'opération suivante:

$$2^s N_{nd,nl} = 2^s(\sum_{i=0}^{i<nl} N_{nd+i}B^i) = (\sum_{i=0}^{i<nl} N'_{nd+i}B^i) + M'_{md}B^{nl} .$$

La fonction **BnShiftLeft** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et le chiffre M_{md} par M'_{md}. Ceci impose donc que $0 \leq s <$ `BN_DIGIT_SIZE`. Le cas particulier $s = 0$ est explicitement testé pour accélérer le temps de calcul.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnShiftLeft n 1 3 n 0 8) n)
= |00000300|00000200|00000100|00000000|
? (progn (BnShiftLeft n 1 3 n 0 16) n)
```

```
= |03000000|02000000|01000000|00000000|
? (progn (BnShiftLeft n 1 3 n 0 16) n)
= |00000200|00000100|00000000|00000300|
```

3.4 Division

(**BnDivideDigit** *Q qd R rd N nd nl M md*)

La fonction **BnDivideDigit** effectue la division du sous-nombre $N_{nd,nl}$ par le chiffre M_{md}, place le quotient dans le sous-nombre $Q_{qd,nl-1}$ et le reste dans le chiffre R_{rd}. Plus formellement l'équation suivante est satisfaite :

$$\sum_{i=0}^{i<nl} N_{nd+i}B^i = (\sum_{i=0}^{i<nl-1} Q_{qd+i}B^i) \times M_{md} + R_{rd} \text{ avec } 0 \leq R_{rd} < M_{md} \ .$$

La fonction **BnDivideDigit** impose donc que $nl > 1$, $N_{nd+nl-1} < M_{md}$ et que $size(Q) \geq qd + nl - 1$.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnDivideDigit n 0 n 2 n 0 3 n 3) n)
= |00000003|00000000|AAAAAAAB|00000000|
? (progn (BnDivideDigit n 0 n 2 n 0 3 n 3) n)
= |00000003|00000000|38E38E39|00000000|
? (progn (BnDivideDigit n 0 n 2 n 0 3 n 3) n)
= |00000003|00000002|12F684BD|AAAAAAAA|
? (progn (BnDivideDigit n 0 n 2 n 1 2 n 0) n)
= |00000003|12F684BF|12F684BD|00000003|
```

(**BnDivide** *N nd nl M md ml*)

La fonction **BnDivide** effectue la division du sous-nombre $N_{nd,nl}$ par le sous-nombre $M_{md,ml}$, place le quotient dans le sous-nombre $N_{nd+ml,nl-ml}$ et le reste dans le sous-nombre $N_{nd,ml}$. Plus formellement l'équation suivante est satisfaite :

$$\sum_{i=0}^{i<nl} N_{nd+i}B^i = (\sum_{i=0}^{i<nl-ml} N'_{nd+ml+i}B^i) \times (\sum_{i=0}^{i<ml} M_{md+i}B^i) + (\sum_{i=0}^{i<ml} N'_{nd+i}B^i)$$

$$\text{avec } 0 \leq \sum_{i=0}^{i<ml} N'_{nd+i}B^i < \sum_{i=0}^{i<ml} M_{md+i}B^i \ .$$

La fonction **BnDivide** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i}. Ceci impose donc que $nl > ml$ et que $N_{nd+nl-1} < M_{md+ml-1}$.

Exemple :

```
? (progn (setq n (BnCreate 'n 5)) (for (i 0 1 4) (BnSetDigit n i i)) n)
= |00000004|00000003|00000002|00000001|00000000|
? (progn (BnDivide n 0 3 n 3 2) n)
= |00000004|00000003|7FFFFFFF|00000003|80000003|
```

```
? (progn (BnDivide n 0 2 n 4 1) n)
= |00000004|00000003|7FFFFFFF|E0000000|00000003|
? (progn (BnDivide n 2 3 n 0 2) n)
= |00000004|80000003|7FFFFFF3|E0000000|00000003|
```

(**BnShiftRight** *N nd nl M md s*)

La fonction **BnShiftRight** décale vers la droite tous les chiffres du sous-nombre $N_{nd,nl}$, les s bits laissés vacants d'un chiffre sont remplacés par les s bits sortants du chiffre suivant, les s bits laissés vacants du dernier chiffre sont remplacés par des zéros, et les s bits sortants du premier chiffre sont placés, cadrés sur les poids forts, dans le chiffre M_{md}. La fonction **BnShiftRight** effectue donc une division par 2^s sur le sous-nombre $N_{nd,nl}$. Plus formellement l'équation suivante est satisfaite :

$$\sum_{i=0}^{i<nl} N_{nd+i}B^i = 2^s(M'_{md}B^{-1} + \sum_{i=0}^{i<nl} N'_{nd+i}B^i) \ .$$

La fonction **BnShiftRight** remplace tous les chiffres N_{nd+i} par leurs équivalents N'_{nd+i} et le chiffre M_{md} par M'_{md}. Ceci impose donc que $0 \leq s <$ **BN_DIGIT_SIZE**. Le cas particulier $s = 0$ est explicitement testé pour accélérer le temps de calcul.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnShiftRight n 1 3 n 0 8) n)
= |00000000|03000000|02000000|01000000|
? (progn (BnShiftRight n 1 3 n 0 16) n)
= |00000000|00000300|00000200|00000000|
? (progn (BnShiftRight n 1 3 n 0 16) n)
= |00000000|00000000|03000000|02000000|
```

Les deux procédures suivantes sont utilisées dans le processus de normalisation de la division longue.

(**BnNumLeadingZeroBitsInDigit** *N nd*)

La fonction **BnNumLeadingZeroBitsInDigit** calcule le nombre de bits nuls dans les poids forts du chiffre N_{nd}. Plus formellement la fonction **BnNumLeadingZeroBitsInDigit** calcule l'entier k tel que l'équation suivante soit satisfaite :

$$\frac{B}{2} < (N_{nd} + 1)2^k \leq B \ .$$

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnNumLeadingZeroBitsInDigit n 0)
= 32
? (BnNumLeadingZeroBitsInDigit n 1)
```

```
= 31
? (BnNumLeadingZeroBitsInDigit n 3)
= 30
? (progn (BnComplement n 2 1) n)
= |00000003|FFFFFFFD|00000001|00000000|
? (BnNumLeadingZeroBitsInDigit n 2)
= 0
```

(**BnIsDigitNormalized** *N nd*)

La fonction **BnIsDigitNormalized** retourne 0 si le chiffre N_{nd} vérifie:

$$0 \leq N_{nd} < \frac{B}{2} .$$

La fonction **BnIsDigitNormalized** teste le bit de poids fort du chiffre N_{nd} et retourne 0 si ce bit a pour valeur 0; dans le cas contraire la fonction retourne un entier différent de 0, la valeur exacte de cet entier peut dépendre des machines.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnIsDigitNormalized n 0)
= 0
? (BnIsDigitNormalized n 1)
= 0
? (progn (BnComplement n 0 2) n)
= |00000003|00000002|FFFFFFFE|FFFFFFFF|
? (BnIsDigitNormalized n 0)
= 255
? (BnIsDigitNormalized n 1)
= 255
```

3.5 Comparaisons

(**BnIsDigitZero** *N nd*)

La fonction **BnIsDigitZero** retourne 0 si le chiffre N_{nd} est non nul.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnIsDigitZero n 0)
= 255
? (BnIsDigitZero n 1)
= 0
```

(**BnIsZero** *N nd nl*)

La fonction **BnIsZero** retourne 0 si le sous-nombre $N_{nd,nl}$ est non nul.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnIsZero n 0 1)
= 255
? (BnIsZero n 0 2)
= 0
? (BnIsZero n 1 1)
= 0
```

(**BnCompareDigits** *N nd M md*)

La fonction **BnCompareDigits** retourne -1 si le chiffre N_{nd} est strictement plus petit que le chiffre M_{md}, 0 si le chiffre N_{nd} est égal au chiffre M_{md} et 1 si le chiffre N_{nd} est strictement supérieur au chiffre M_{md}.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnCompareDigits n 0 n 0)
= 0
? (BnCompareDigits n 0 n 1)
= -1
? (BnCompareDigits n 1 n 0)
= 1
? (progn (BnComplement n 0 2) n)
= |00000003|00000002|FFFFFFFE|FFFFFFFF|
? (BnCompareDigits n 0 n 0)
= 0
? (BnCompareDigits n 0 n 1)
= 1
? (BnCompareDigits n 1 n 0)
= -1
```

(**BnCompare** *N nd nl M md ml*)

La fonction **BnCompare** retourne -1 si le sous-nombre $N_{nd,nl}$ est strictement plus petit que le sous-nombre $M_{md,ml}$, retourne 0 si le sous-nombre $N_{nd,nl}$ est égal au sous-nombre $M_{md,ml}$ et retourne 1 si le sous-nombre $N_{nd,nl}$ est strictement supérieur au sous-nombre $M_{md,ml}$.

Exemple :

```
? (progn (setq n (BnCreate 'n 5)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000000|00000003|00000002|00000001|00000000|
? (BnCompare n 2 3 n 2 2)
= 0
? (BnCompare n 2 3 n 0 2)
```

```
= 1
? (BnCompare n 3 2 n 0 2)
= -1
? (progn (BnComplement n 0 2) n)
= |00000000|00000003|00000002|FFFFFFFE|FFFFFFFF|
? (BnCompare n 0 2 n 0 2)
= 0
? (BnCompare n 0 1 n 1 1)
= 1
? (BnCompare n 1 1 n 0 1)
= -1
? (BnCompare n 2 3 n 0 2)
= -1
? (BnCompare n 3 2 n 0 2)
= -1
```

(BnIsDigitOdd *N nd*)

La fonction **BnIsDigitOdd** retourne la valeur 0 si le chiffre N_{nd} est pair. Cette fonction test le bit de poids faible du chiffre N_{nd}.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnIsDigitOdd n 0)
= 0
? (BnIsDigitOdd n 1)
= 255
? (progn (BnComplement n 0 2) n)
= |00000003|00000002|FFFFFFFE|FFFFFFFF|
? (BnIsDigitOdd n 0)
= 255
? (BnIsDigitOdd n 1)
= 0
```

(BnNumDigits *N nd nl*)

La fonction **BnNumDigits** retourne le nombre de chiffres significatifs du sous-nombre $N_{nd,nl}$. Cette fonction retourne la valeur 1 pour le cas particulier où le sous-nombre $N_{nd,nl}$ vaut 0.

Exemple:

```
? (setq n (BnCreate 'n 4))
= |00000000|00000000|00000000|00000000|
? (BnNumDigits n 0 4)
= 1
? (progn (BnSetDigit n 1 3) n)
= |00000000|00000000|00000003|00000000|
? (BnNumDigits n 0 3)
```

```
= 2
? (progn (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnNumDigits n 0 4)
= 4
? (BnNumDigits n 2 0)
= 1
```

3.6 Fonctions logiques

(**BnAndDigits** *N nd N md*)

La fonction **BnAndDigits** effectue le **et logique** entre les chiffres N_{nd} et M_{md}, le chiffre résultant est mis à la place du chiffre N_{nd}.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnAndDigits n 3 n 1) n)
= |00000001|00000002|00000001|00000000|
? (progn (BnAndDigits n 0 n 2) n)
= |00000001|00000002|00000001|00000000|
```

(**BnOrDigits** *N nd M md*)

La fonction **BnOrDigits** effectue le **ou logique** entre les chiffres N_{nd} et M_{md}, le chiffre résultant est mis à la place du chiffre N_{nd}.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnOrDigits n 3 n 1) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnOrDigits n 0 n 2) n)
= |00000003|00000002|00000001|00000002|
```

(**BnXorDigits** *N nd M md*)

La fonction **BnXorDigits** effectue le **ou exclusif logique** entre les chiffres N_{nd} et M_{md}, le chiffre résultant est mis à la place du chiffre N_{nd}.

Exemple :

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (progn (BnXorDigits n 3 n 1) n)
```

```
= |00000002|00000002|00000001|00000000|
? (progn (BnXorDigits n 0 n 2) n)
= |00000002|00000002|00000001|00000002|
```

3.7 Affectations

Les fonctions suivantes permettent de manipuler directement la représentation interne des nombres.

(**BnSetToZero** *N nd nl*)

La fonction **BnSetToZero** remet à zéro le sous-nombre $N_{nd,nl}$.

Exemple :

```
? (progn (setq n (BnCreate 'n 7)) (for (i 0 1 6) (BnSetDigit n i i)) n)
= |00000006|00000005|00000004|00000003|00000002|00000001|00000000|
? (progn (BnSetToZero n 2 4) n)
= |00000006|00000000|00000000|00000000|00000000|00000001|00000000|
? (progn (BnSetToZero n 1 0) n)
= |00000006|00000000|00000000|00000000|00000000|00000001|00000000|
```

(**BnSetDigit** *N nd digit*)

La fonction **BnSetDigit** permet d'affecter le chiffre N_{nd} à la valeur entière *digit*.

Exemple :

```
? (setq n (BnCreate 'n 4))
= |00000000|00000000|00000000|00000000|
? (progn (BnSetDigit n 3 12) n)
= |0000000C|00000000|00000000|00000000|
? (progn (BnSetDigit n 0 255) n)
= |0000000C|00000000|00000000|000000FF|
```

(**BnAssign** *M md N nd nl*)

La fonction **BnAssign** transfert le sous-nombre $N_{nd,nl}$ dans le sous-nombre $M_{md,nl}$ avec perte de l'ancien sous-nombre $M_{md,nl}$. Le chevauchement des zones de transfert est valide. Pour que le sous-nombre $M_{md,nl}$ soit valide, la condition suivante doit être vérifiée :

$$md + nl \leq size(M) .$$

Exemple :

```
? (progn (setq n (BnCreate 'n 7)) (for (i 0 1 6) (BnSetDigit n i i)) n)
= |00000006|00000005|00000004|00000003|00000002|00000001|00000000|
```

```
? (progn (setq m  (BnCreate 'n 9))
?        (for (i 0 1 8) (BnSetDigit m i (+ 10 i)))
?        m)
= |00000012|00000011|00000010|0000000F|0000000E|0000000D|0000000C|0000000B|0000000A|
? (progn (BnAssign m 1 n 2 4) m)
= |00000012|00000011|00000010|0000000F|00000005|00000004|00000003|00000002|0000000A|
? (progn (BnAssign m 0 m 1 3) m)
= |00000012|00000011|00000010|0000000F|00000005|00000004|00000004|00000003|00000002|
? (progn (BnAssign m 3 m 0 6) m)
= |0000000F|00000005|00000004|00000004|00000003|00000002|00000004|00000003|00000002|
? (progn (BnAssign m 3 m 0 0) m)
= |0000000F|00000005|00000004|00000004|00000003|00000002|00000004|00000003|00000002|
```

3.8 Conversion en fix Le-Lisp

En Le-Lisp, il n'est pas forcement possible de définir, comme un simple entier, l'intervalle représentant les chiffres (i.e. les entiers compris entre 0 et B). Le prédicat suivant spécifie quels sont les chiffres pouvant être représentés par entier Le-Lisp. La taille, en nombre de bits, des chiffres au delà de laquelle une conversion vers un entier Le-Lisp est possible est définie par la constante: **BN_WORD_SIZE**. Cette constante vaut 15 pour la plupart des machines. L'implantation BigNum.c suppose implicitement que les constantes **BN_WORD_SIZE** et **BN_DIGIT_SIZE** ont la même valeur. Cette restriction permet de passer directement les chiffres en paramètres des primitives (au lieu d'utiliser deux variables N et nd pour le chiffre N_{nd}).

(**BnDoesDigitFitInWord** N nd)

La fonction **BnDoesDigitFitInWord** retourne 0 si le chiffre N_{nd} ne peut pas être converti dans un entier Le-Lisp.

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnDoesDigitFitInWord n 2)
= 1
? (progn (BnComplement n 0 3) n)
= |00000003|FFFFFFFD|FFFFFFFE|FFFFFFFF|
? (BnDoesDigitFitInWord n 2)
= 0
```

(**BnGetDigit** n nd)

La fonction **BnGetDigit** retourne le chiffre N_{nd}. Il n'est pas vérifié que ce dernier puisse être représenté par un entier Le-Lisp (se référer à la fonction **BnDoesDigitFitInWord**).

Exemple:

```
? (progn (setq n (BnCreate 'n 4)) (for (i 0 1 3) (BnSetDigit n i i)) n)
= |00000003|00000002|00000001|00000000|
? (BnGetDigit n 2)
```

```
= 2
? (progn (BnComplement n 1 2) n)
= |00000003|FFFFFFFD|FFFFFFFE|00000000|
? (BnGetDigit n 2)
= -3.595385e+N8
```

3.9 Création et gestion

Les fonctions d'ordre plus général, du module **Bn** sont:

(**BnAlloc** *size*)

La fonction **BnAlloc** alloue la place pour un nombre pouvant contenir au plus *size* chiffres. On impose que *size* soit un nombre *strictement positif*. Les valeurs des chiffres du nombre rendu en résultat par la fonction **BnAlloc** sont imprévisibles.

(**BnCreate** *type size*)

La fonction **BnCreate** alloue la place pour un nombre pouvant contenir au plus *size* chiffres, initialise le type de ce nombre à *type* et initialise tous les chiffres du nombre à 0. On impose que *size* soit un nombre *strictement positif*.

Exemple:

```
? (BnCreate 'n 4)
= |00000000|00000000|00000000|00000000|
```

(**BnSetType** *N type*)

La fonction **BnSetType** permet de modifier le champ *type* du nombre N. La spécification de **KerN** n'utilise pas pour ses besoins propres le type d'un nombre, mais ce champ donne la possibilité de définir, dans des librairies de plus haut niveau, des fonctions génériques, c'est à dire des fonctions ayant des comportements différents selon le type de ses arguments. Par exemple le champ *type* peut contenir le signe d'un nombre.

Exemple:

```
? (progn (BnSetType (BnAlloc 8) 'bar) 'ok)
= ok
```

(**BnGetType** *N*)

La fonction **BnGetType** retourne le type associé au nombre N.

Exemple:

```
? (progn (BnSetType (setq n (BnAlloc 8)) 'foo) (BnGetType n))
```

```
= foo
? (progn (BnSetType n 'bar) (BnGetType n))
= bar
```

(BnGetSize *N*)

La fonction **BnGetSize** retourne le nombre de chiffres disponibles dans le nombre N. Il n'est pas possible de changer dynamiquement la taille d'un nombre; ainsi, pour un même nombre, **BnGetSize** retourne toujours la même valeur.

Exemple:

```
? (BnGetSize (BnAlloc 8))
= 8
```

4 Allocation des résultats : Bz

Le niveau **Bz** est conceptuellement plus simple que celui de **Bn**. Un nombre $z \in Z$ est représenté par un BigNum dont le champ type contient le signe de l'entier. Les fonctions de **Bz** sont sans effet de bord, et allouent l'espace mémoire pour leurs résultats. Toutes les fonctions de ce niveau sont directement interfacées avec l'arithmétique générique de Le-Lisp.

4.1 Fonctions arithmétiques

(BzAbs *Z*)

La fonction **BzAbs** retourne la valeur absolue du nombre Z.

Exemple:

```
? (BzAbs 1234567890123)
= 1234567890123
? (BzAbs -1234567890123)
= 1234567890123
? (BzAbs #{0})
= 0
```

(BzSign *Z*)

La fonction **BzSign** retourne -1 si le nombre Z est strictement négatif, 0 si ce nombre est nul et 1 si ce nombre est strictement positif.

Exemple:

```
? (BzSign 1234567890123)
= 1
? (BzSign -1234567890123)
```

```
= -1
? (BzSign #{0})
= 0
```

(BzNegate *Z*)

La fonction **BzNegate** retourne l'opposé du nombre *Z*.

Exemple:

```
? (BzNegate 1234567890123)
= -1234567890123
? (BzNegate -1234567890123)
= 1234567890123
? (BzNegate #{0})
= 0
```

(BzCompare *Y Z*)

La fonction BzCompare retourne -1 si le nombre *Y* est strictement plus petit que le nombre *Z*, 0 si ces deux nombres sont égaux et 1 si le nombre *Y* et strictement plus grand que le nombre *Z*.

Exemple:

```
? (BzCompare 1234567890123 1234567890123)
= 0
? (BzCompare 1234567890123 123456789)
= 1
? (BzCompare 1234567890123 -1234567890123)
= 1
? (BzCompare -1234567890123 -123456789)
= -1
```

(BzAdd *Y Z*)

La fonction **BzAdd** retourne un nombre représentant la somme des nombres *Y* et *Z*.

Exemple:

```
? (BzAdd 1234567890123 123456789)
= 1234691346912
? (BzAdd 1234567890123 -1234567890123)
= 0
? (BzAdd -1234567890123 -123456789)
= -1234691346912
```

(BzSubtract *Y Z*)

La fonction **BzSubtract** retourne un nombre représentant la différence des nombres Y et Z.

Exemple:

```
? (BzSubtract 1234567890123 123456789)
= 1234444433334
? (BzSubtract 1234567890123 -1234567890123)
= 2469135780246
? (BzSubtract -1234567890123 -123456789)
= -1234444433334
```

(BzMultiply Y Z)

La fonction **BzMultiply** retourne un nombre représentant le produit des nombres Y et Z.

Exemple:

```
? (BzMultiply 1234567890123 123456789)
= 152415787517090395047
? (BzMultiply 1234567890123 -1234567890123)
= -152415787532275800955129
? (BzMultiply -1234567890123 -123456789)
= 152415787517090395047
```

(BzDivide Y Z)

La fonction **BzDivide** effectue la division du nombre Y par le nombre Z et en retourne le quotient.

Exemple:

```
? (BzDivide 1234567890123 123456789)
= 10000
? (BzDivide 1234567890123 -1234567890123)
= -1
? (BzDivide -1234567890123 -123456789)
= 10001
```

(BzMod Y Z)

La fonction **BzMod** effectue la division du nombre Y par le nombre Z et en retourne le reste.

Exemple:

```
? (BzMod 1234567890123 123456789)
= 123
? (BzMod 1234567890123 -1234567890123)
= 0
? (BzMod -1234567890123 -123456789)
= 123456666
```

4.2 Création

(**BzCreate** *size*)

La fonction **BzCreate** alloue la place pour un nombre pouvant contenir au plus *size* chiffres et initialise tous les chiffres du nombre à 0. On impose que *size* soit un nombre *strictement positif*.

Exemple:

```
? (BzCreate 4)
= 0
```

(**BzCopy** *Z*)

La fonction **BzCopy** retourne une copie du nombre *Z*.

Exemple:

```
? (BzCopy 1234567890123)
= 1234567890123
? (BzCopy -1234567890123)
= -1234567890123
? (BzCopy #{0})
= 0
```

4.3 Lecture et écriture

Les nombres de BigNum peuvent être lus et écrits comme les petits entiers Le-Lisp. Néanmoins certaines restrictions du système d'entrée-sortie de Le-Lisp nous ont contraint à introduire la notation #{*Digit**}. Les restrictions sont les suivantes:

1. Les nombres de BigNum ne peuvent pas être lus de manière standard lorsque la base des nombres en entrée est différente de dix.
2. L'impression d'un nombre comportant trop[4] de chiffres peut faire apparaître des caractères parasites de fin de ligne. De plus, du fait de la présence de ces caractères parasites, la relecture d'un nombre, à partir de sa représentation externe, peut provoquer des erreurs.

La notation #{*Digit**} permet:

1. De lire un nombre dans la base en entrée courante.
2. D'ignorer les caractères ne faisant pas parti des chiffres admissibles par la base en entrée courante. Ceci permet, entre autre, de scinder les grands nombres en plusieurs lignes.
3. D'imprimer, lorsque la variable Le-Lisp **#:system:print-for-read** est positionnée, des nombres qui seront ultérieurement relisible.

[4] Le *trop* est dépendant de la valeur de la marge droite et de la taille maximum d'un symbole

Voici une petite session mettant en jeu les différents points énoncés ci-dessus :

```
? 2934809238409238450679562634 5          ; lecture en standard.
= 2934809238409238450679562634 5
? (ibase 16)                              ; on change de base.
= 16
? add                                     ; Pour les petits ca passe.
= 2781
? decede                                  ; Pour les gros ca casse.
= -12578
? #{decede}                               ; En fait il faut lire.
= 14601950
? (ibase a)                               ; Repasse en base 10.
= 10
? (defun foo ()
?    ; Ecrit et relit le meme nombre.
?    (with ((outchan (openo "/tmp/foo")))
?       (print (** (** (** (** 2 3) 4 ) 5) 6))
?       (close (outchan)) )
?    (with ((inchan (openi "/tmp/foo")))
?       (untilexit eof (print (read)) (print "another")) ))
= foo
? (foo)
234854258277383322788948059678933702737568254890831987070729097153220902511460
another
844346369899838476870303193 4976
another
= 8
? (setq #:system:print-for-read t)
= t
? (foo)
#{234854258277383322788948059678933702737568254890831987070729097153220902511 4
6084434636989983847687030319349 6}
"another"
= 8
? #{123 456 ;on met des espaces tous les 3 caracteres
? 789
? }
= #{1234563789}
? (obase (ibase 16))
= 16
? #{123 456 ;on met des espaces tous les 3 caracteres
? 789
? }
= #{123456EDEEACEE3CAACEE789}
```

5 Remerciements

Patrice Bertin, Hans Boehm et Michel Gangnet ont contribué à la réalisation de ce module. Qu'ils en soient remerciés ici.

6 Bibliography

[Chailloux 87] Jérôme Chailloux, Matthieu Devin, Francis Dupont, Jean-Marie Hullot, Bernard Serpette, Jean Vuillemin. *Le-Lisp version 15.2, le Manuel de référence.* Documentation INRIA, Mai 1987.

[Knuth] D. E. Knuth, *The Art of Computer Programming, vol. 2, Seminumerical Algorithms.* Addison Wesley, 1981.

7 Distribution du module

Ce document, et les sources du module BigNum porte la mention "Copyright Digital Equipment Corporation & INRIA 1988, 1989". Cette documentation et les sources du module BigNum peuvent être obtenus, en contactant par courrier électronique ou postal,:

```
Digital P.R.L.,
Attn Librarian,
85, Avenue Victor Hugo,
92563 Rueil Malmaison Cedex. France.          (adresse postale)

librarian@decprl.dec.com                      (adresse electronique)
```

ou,:

```
I.N.R.I.A.,
A l'attention de B. Serpette,
Domaine de Voluceau,
78150, Rocquencourt, France.                 (adresse postale)

serpette@inria.inria.fr                      (adresse electronique)
```

La documentation et les sources du module BigNum peuvent être librement reproduites et distribuées moyennant les conditions suivantes:

- Digital PRL ou l'INRIA doivent être avertis de la copie.
- La mention du *copyright* ne doit en aucun cas être enlevée de la documentation et des sources.
- Tout travail incorporant le module BigNum devra mentionner explicitement l'utilisation de ce module et son origine en incluant cette phrase: *Ce travail utilise le module BigNum développé conjointement par l'INRIA et Digital PRL.*
- Toutes les modifications effectuées sur le module BigNum devront être explicitement notifiées par la nature de la modification, son auteur et son adresse. Ces notifications ne pourront être enlevées des distributions futures.
- Tout travail utilisant intensivement le module BigNum peut être librement distribué sous les mêmes conditions de distribution que le module BigNum.

L'INRIA et Digital Equipment Corporation n'offrent aucune garantie, implicite ou explicite, sur cette documentation et sur le logiciel qu'elle décrit, en ce qui concerne la diffusabilité ou l'adéquation à une utilisation particulière. l'INRIA et Digital Equipment Corporation ainsi que les distributeurs à venir ne seront en aucun cas responsables des dommages produits.

Contents

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique