Numerix

Bibliothèque de calcul en grands entiers

Michel Quercia
michel.quercia@prepas.org

version 0.19 du 24 août 2001

Table des matières

Chapitre 1

Présentation

Numerix est une bibliothèque implémentant les nombres entiers relatifs de longueur arbitraire et les principales opérations arithmétiques sur ces nombres. Conçue pour être utilisée avec le langage Objective-Caml, elle est aussi disponible avec des fonctionnalités réduites pour les langages Camllight, C et Pascal sur des machines de type Unix 32 ou 64 bits. Elle est fournie sous trois versions :

`Clong` :
: écrite en C standard. L'entité de base est le « chiffre » dont la longueur en bits est la moitié de celle d'un mot machine. De la sorte les opérations élémentaires produisant des résultats sur deux chiffres sont implémentées par des opérations ordinaires du C sur des variables de type `long`, et cette bibliothèque est à priori portable sur toute architecture binaire pour laquelle la taille d'un mot est paire et d'au moins 32 bits.

`Dlong` :
: écrite en C aussi, mais un chiffre occupe un mot machine en entier. Les opérations entre chiffres sont réalisées avec le type `long long` de `gcc` ce qui limite à l'heure actuelle cette bibliothèque aux machines 32 bits (sur les machines 64 bits de l'Inria, un `long long` a la même taille qu'un `long` et on ne dispose pas d'opérations double précision).

`Slong` :
: écrite pour partie en assembleur pour processeurs Intel x86 et pour partie en C, un chiffre est un mot de 32 bits.

En termes de performances, Numerix se compare favorablement aux autres bibliothèques de calcul multiprécision couramment disponibles en particulier Big_int (adaptation pour Camllight/Ocaml de BigNum) et GMP. A titre d'exemple voici les temps de calcul des n premières décimales de π sur un PC-Linux équipé d'un processeur Athlon-550Mhz avec 128Mo de mémoire vive :

n	Slong	Dlong	Clong	GMP-3	Big_int
10^4	0.06s	0.24s	0.33s	0.08s	0.45s
10^5	1.49s	6.49s	9.05s	4.95s	57.22s
10^6	36.27s	148.71s	208.61s	505.58s	15867s

Dans tous les cas le même algorithme est utilisé, déduit d'un développement en série de Ramanujan, seule l'implémentation des grands entiers change. Les bibliothèques `Slong`, `Clong`, `Dlong` et `GMP` ont été utilisées avec un programme principal en C, `Big_int` avec un programme principal en Ocaml. L'incidence du programme principal sur le temps d'exécution est toutefois négligeable pour ce type de programmes où la majeure partie du temps est passée dans les calculs sur des nombres de plusieurs millions de bits ; on observe des temps similaires lorsque les cinq bibliothèques sont pilotées par un programme principal en Ocaml.

Voici les temps de calcul sur une station Dec avec un processeur alpha-500 et 256Mo de mémoire vive :

n	`Clong`	`Big_int`
10^4	0.15s	0.55s
10^5	4.68s	60s
10^6	112s	8310s

La clé de cette grande vitesse de calcul réside dans l'implémentation d'algorithmes rapides pour toutes les opérations, notamment la multiplication (algorithmes de Karatsuba, et Schönage (FFT)) la division (algorithme de Burnikel-Ziegler), la racine carrée (algorithme de Zimmermann), le pgcd (algorithme de Lehmer récursif) et la conversion en décimal. `GMP-3.1` implémente des algorithmes similaires pour la multiplication et la division ce qui lui confère pour ces opérations une vitesse meilleure que celle de `Numerix` avec des opérandes de taille moyenne, moindre avec des opérandes dépassant le million de bits. Mais les autres opérations sont implémentées par des algorithmes de complexité quadratique, ce qui explique un temps total d'exécution plus élevé en général avec `GMP` qu'avec `Numerix`. De plus l'interfaçage de `GMP` avec Camllight ou avec Ocaml conduit à une gestion de la mémoire inefficace pouvant dans certains cas provoquer un ralentissement supplémentaire voire l'arrêt du programme pour manque de mémoire. Quant à `Big_int`, toutes les opérations à partir de la multiplication ont une complexité au moins quadratique, donc elle n'est pas adaptée aux calculs sur des nombres de grande taille.

De l'autre côté de l'échelle, pour des calculs sur des « petits » nombres ne dépassant pas 300 bits, `Numerix` affiche également de bonnes performances face à `Big_int` pour des raisons de moindre charge imposée au gestionnaire mémoire et de prise en compte au niveau C ou assembleur des signes et longueurs des entiers manipulés au lieu de confier cette gestion au code Caml/Ocaml. `GMP`, bien que disposant d'algorithmes optimisés pour des opérandes de petite taille, se révèle généralement moins rapide que `Numerix` dans son utilisation par un programme en Caml ou en Ocaml du fait d'une mauvaise adéquation entre les gestionnaires mémoire intégrés à `GMP` d'une part et à Caml/Ocaml d'autre part. Le problème de Hamming (trouver le n-ème entier naturel n'ayant que 2,3 et 5 comme diviseurs premiers) conduit à effectuer de l'ordre de $n^{1/3}\ln(n)$ opérations sur des entiers de petite taille, entre 100 et 300 bits. Voici les temps de calcul avec chacune des bibliothèques interfacées à Ocaml (sur PC-Linux Athlon-550, 128Mo de RAM) :

avec Ocaml 2.04

n	Slong	Dlong	Clong	GMP-3	Big_int
10^{12}	0.17s	0.28s	0.26s	2.13s	0.69s
10^{15}	1.51s	2.52s	2.44s	20.23s	6.80s
10^{18}	17.06s	29.07s	28.99s	233.62s	79.59s
10^{21}	214.33s	355.56s	370.77s	2815.57s	891.14s

avec Ocaml 3.00

n	Slong	Dlong	Clong	GMP-3	Big_int
10^{12}	0.17s	0.27s	0.26s	2.54s	0.70s
10^{15}	1.47s	2.61s	2.57s	–	6.58s
10^{18}	17.05s	30.10s	30.77s	–	75.52s
10^{21}	213.15s	368.21s	386.83s	–	933.48s

`GMP` ne passe pas ce test pour $n \geqslant 10^{15}$ avec Ocaml-3.00 et provoque un débordement mémoire.

Bien que conçue pour être interfacée avec les langages fonctionnels Camllight et Ocaml, `Numerix` offre la possibilité d'effectuer les opérations « sur place », c'est à dire en utilisant la mémoire attribuée à une variable pour stocker le résultat d'une opération au lieu d'allouer un nouveau bloc mémoire pour ce résultat. Si le bloc mémoire attribué à une variable se révèle insuffisant pour contenir le résultat à stocker, alors un nouveau bloc de taille convenable est alloué à cette variable et l'ancien est abandonné aux bons soins du gestionnaire mémoire de Camllight/Ocaml pour récupération éventuelle. L'accroissement de taille se fait selon une politique de « doublement des besoins » (on alloue un bloc deux fois plus gros que nécessaire) de façon à éviter des réallocations continuelles. Le calcul sur place avec cette politique de doublement présente l'avantage de réduire pratiquement à zéro la charge du gestionnaire mémoire et son influence sur le temps total d'exécution d'un programme, au prix d'une programmation moins intuitive et d'une consommation mémoire accrue. Voici les temps de calcul pour le problème de Hamming en version surplace :

avec Ocaml 2.04

n	Slong	Dlong	Clong	GMP-3	Big_int
10^{12}	0.15s	0.27s	0.26s	0.85s	0.69s
10^{15}	1.33s	2.39s	2.26s	3.24s	6.74s
10^{18}	15.20s	27.35s	26.31s	23.10s	78.77s
10^{21}	187.06s	334.55s	330.44s	285.82s	978.26s

avec Ocaml 3.00

n	Slong	Dlong	Clong	GMP-3	Big_int
10^{12}	0.17s	0.28s	0.27s	0.94s	0.76s
10^{15}	1.38s	2.58s	2.36s	3.49s	7.51s
10^{18}	15.46s	29.56s	27.30s	22.96s	87.15s
10^{21}	189.25s	360.63s	343.15s	284.75s	1074.41s

La bibliothèque `Big_int` est ici désavantagée par ce test car les opérations sur place n'y sont pas disponibles et elles ont été simulées par l'usage de références ce qui augmente le temps d'exécution par rapport à la version fonctionnelle. Par contre, les problèmes de gestion mémoire entre `GMP` et Ocaml sont éliminés par cette programmation sur place.

Chapitre 2

Utilisation avec Ocaml

2.1 Interface

`Numerix` a été développé et testé avec Ocaml-2.02 puis avec Ocaml-3.00. Quelques essais avec d'autres versions de Ocaml n'ont pas posé de problème particulier à l'exception de Ocaml-1.05 pour lequel l'activation automatique des pretty-printers dans `ocamlnumx` n'a pas pu être implémentée (mais l'activation manuelle par `install_printer` fonctionne). Il est donc probable que `Numerix` fonctionne correctement pour toute version de Ocaml comprise au sens large entre 1.05 et 3.02.

Les cinq bibliothèques `Clong`, `Dlong`, `Slong`, `GMP` et `Big_int` sont déclarées comme des sous-modules du module `Numerix` avec la même signature de type `Int_type` dans laquelle les types de données grand entier et grand entier mutable (pour les calculs sur place) sont abstraits. Ceci permet d'écrire un code utilisateur indépendant d'une implémentation particulière des grands entiers et de tester ce code avec les diverses implémentations disponibles sans avoir à recompiler le programme, grâce au système de modules de Ocaml et à la possibilité de définir des modules paramétrés (foncteurs). Voici l'interface publique extraite du fichier `numerix.mli` :

```
module type Int_type = sig
type t      (* entier abstrait          *)
type tref   (* entier mutable abstrait *)

(* a,b,c,... désignent les arguments des fonctions dans cet ordre *)
(* opérations unaires et binaires *)
val add         : t → t → t             (* a + b                *)
val sub         : t → t → t             (* a − b                *)
val mul         : t → t → t             (* a ∗ b                *)
val quo         : t → t → t             (* a div b              *)
val modulo      : t → t → t             (* a mod b              *)
val quomod      : t → t → t*t           (* (a div b, a mod b)   *)
val abs         : t → t                 (* |a|                  *)
val neg         : t → t                 (* −a                   *)
val sqr         : t → t                 (* a^2                  *)
```

```
val sqrt          : t → t                   (* ⌊√a⌋                                     *)
val powmod        : t → t → t → t           (* a^b mod c, b ⩾ 0                         *)
val gcd           : t → t → t               (* pgcd(a,b)                                *)
val gcd_ex        : t → t → t*t*t           (* coefficients de Bézout et pgcd           *)
val cfrac         : t → t → t*t*t*t*t       (* entiers réduits, coef. de Bézout et pgcd *)
val eq            : t → t → bool            (* a = b ?                                  *)
val neq           : t → t → bool            (* a ≠ b ?                                  *)
val inf           : t → t → bool            (* a < b ?                                  *)
val infeq         : t → t → bool            (* a ⩽ b ?                                  *)
val sup           : t → t → bool            (* a > b ?                                  *)
val supeq         : t → t → bool            (* a ⩾ b ?                                  *)
val sgn           : t → int                 (* signe(a)     (1,−1,0)                    *)
val cmp           : t → t → int             (* signe(a − b) (1,−1,0)                    *)
val max           : t → t → t               (* max(a,b)                                 *)
val min           : t → t → t               (* min(a,b)                                 *)

(* opérations mixtes *)
val add_1         : t → int → t             (* a + b                                    *)
val sub_1         : t → int → t             (* a − b                                    *)
val mul_1         : t → int → t             (* a ∗ b                                    *)
val quo_1         : t → int → t             (* a div b                                  *)
val mod_1         : t → int → int           (* a mod b                                  *)
val quomod_1      : t → int → t*int         (* (a div b, a mod b)                       *)
val eq_1          : t → int → bool          (* a = b ?                                  *)
val neq_1         : t → int → bool          (* a ≠ b ?                                  *)
val inf_1         : t → int → bool          (* a < b ?                                  *)
val infeq_1       : t → int → bool          (* a ⩽ b ?                                  *)
val sup_1         : t → int → bool          (* a > b ?                                  *)
val supeq_1       : t → int → bool          (* a ⩾ b ?                                  *)
val cmp_1         : t → int → int           (* signe(a − b) (1,−1,0)                    *)
val pow           : t → int → t             (* a^b                                      *)
val root          : t → int → t             (* ⌊ᵇ√a⌋                                    *)
val fact          : int → t                 (* a!                                       *)
val random_bits   : int → t                 (* nombre pseudo-aléatoire                  *)

(* accès à la représentation binaire *)
val nbits         : t → int                 (* taille en bits de |a|                    *)
val lowbits       : t → int                 (* bits 0..30 de |a|                        *)
val highbits      : t → int                 (* bits n − 31..n − 1, n =nbits(a)          *)
val nth_bit       : t → int → bool          (* bit b de |a|                             *)
val nth_word      : t → int → int           (* bits 16b..16b + 15 de |a|                *)
val shr           : t → int → t             (* sgn(a) ∗ (|a| div 2^b)                   *)
val shl           : t → int → t             (* a ∗ 2^b                                  *)
val split         : t → int → t*t           (* sgn(a) ∗ (|a| div 2^b, |a| mod 2^b)      *)
val join          : t → t → int → t         (* a + b ∗ 2^c                              *)

(* fonctions de conversion *)
val of_int        : int → t
val of_string     : string → t
val int_of        : t → int
```

```
val string_of     : t → string

(* opérations sur place *)
val add_in        : tref → t → t → unit          (* a ← b + c                      *)
val sub_in        : tref → t → t → unit          (* a ← b − c                      *)
val mul_in        : tref → t → t → unit          (* a ← b ∗ c                      *)
val quo_in        : tref → t → t → unit          (* a ← b div c                    *)
val quomod_in     : tref → tref → t → t → unit   (* division et reste              *)
val abs_in        : tref → t → unit              (* a ← |b|                        *)
val neg_in        : tref → t → unit              (* a ← −b                         *)
val sqr_in        : tref → t → unit              (* a ← b^2                        *)
val sqrt_in       : tref → t → unit              (* a ← ⌊√b⌋                       *)
val powmod_in     : tref → t → t → t → unit      (* a ← b^c mod d, c ⩾ 0           *)
val add_1_in      : tref → t → int → unit        (* a ← b + c                      *)
val sub_1_in      : tref → t → int → unit        (* a ← b − c                      *)
val mul_1_in      : tref → t → int → unit        (* a ← b ∗ c                      *)
val quomod_1_in   : tref → t → int → int         (* a ← b div c, retourne b mod c  *)
val pow_in        : tref → t → int → unit        (* a ← b^c                        *)
val root_in       : tref → t → int → unit        (* a ← ⌊ᶜ√b⌋                      *)
val fact_in       : tref → int → unit            (* a ← b!                         *)
val shl_in        : tref → t → int → unit        (* a ← b shl c                    *)
val shr_in        : tref → t → int → unit        (* a ← b shr c                    *)
val split_in      : tref→tref→t→int→unit         (* (a, b) ← c split d             *)
val join_in       : tref → t → t → int → unit    (* a ← b + c ∗ 2^d                *)
val gcd_in        : tref → t → t → unit          (* a ← pgcd(b, c)                 *)
val gcd_ex_in     : tref→tref→tref→t→t→unit      (* coef. de Bézout et pgcd        *)
val cfrac_in      : tref → tref → tref → tref → tref → t → t → unit
                                   (* entiers réduits, coef. de Bézout et pgcd *)

(* gestion des références *)
val make_ref      : t → tref                     (* ref(copy a)                    *)
val copy_in       : tref → t → unit              (* a ← b                          *)
val copy_out      : tref → t                     (* copy(!a)                       *)
val look          : tref → t                     (* !a                             *)

(* divers *)
val name          : unit -> string               (* nom du module                  *)
val toplevel_print : t → unit                    (* affichage avec Format          *)

end (* module type Int_type *)

                          (* notation infixe *)

module Infixes(E : Int_type) : sig
open E

val ( + )         : t → t → t                    (* add                            *)
val ( - )         : t → t → t                    (* sub                            *)
val ( * )         : t → t → t                    (* mul                            *)
val ( / )         : t → t → t                    (* quo                            *)
```

```
val ( % )        : t → t → t               (* modulo                     *)
val ( /% )       : t → t → t*t             (* quomod                     *)
val ( << )       : t → int → t             (* shl                        *)
val ( >> )       : t → int → t             (* shr                        *)
val ( ** )       : t → int → t             (* pow                        *)

val ( +. )       : t → int → t             (* add_1                      *)
val ( -. )       : t → int → t             (* sub_1                      *)
val ( *. )       : t → int → t             (* mul_1                      *)
val ( /. )       : t → int → t             (* quo_1                      *)
val ( %.)        : t → int → int           (* mod_1                      *)
val ( /%.)       : t → int → t*int         (* quomod_1                   *)

val ( +! )       : int → int → int         (* add                        *)
val ( -! )       : int → int → int         (* sub                        *)
val ( *! )       : int → int → int         (* mul                        *)
val ( /! )       : int → int → int         (* quo                        *)
val ( %!)        : int → int → int         (* mod                        *)

val ( = )        : t → t → bool            (* eq (Ocaml < 3.00)          *)
val ( <> )       : t → t → bool            (* neq (Ocaml < 3.00)         *)
val ( < )        : t → t → bool            (* inf (Ocaml < 3.00)         *)
val ( <= )       : t → t → bool            (* infeq (Ocaml < 3.00)       *)
val ( > )        : t → t → bool            (* sup (Ocaml < 3.00)         *)
val ( >= )       : t → t → bool            (* supeq (Ocaml < 3.00)       *)

val ( =. )       : t → int → bool          (* eq_1                       *)
val ( <>. )      : t → int → bool          (* neq_1                      *)
val ( <. )       : t → int → bool          (* inf_1                      *)
val ( <=. )      : t → int → bool          (* infeq_1                    *)
val ( >. )       : t → int → bool          (* sup_1                      *)
val ( >=. )      : t → int → bool          (* supeq_1                    *)

val ( =! )       : 'a → 'a → bool          (* eq                         *)
val ( <>! )      : 'a → 'a → bool          (* neq                        *)
val ( <! )       : 'a → 'a → bool          (* inf                        *)
val ( <=! )      : 'a → 'a → bool          (* infeq                      *)
val ( >! )       : 'a → 'a → bool          (* sup                        *)
val ( >=! )      : 'a → 'a → bool          (* supeq                      *)

val ( ~ )        : tref → t                (* look (Ocaml < 3.00)        *)
val ( ~~ )       : tref → t                (* look                       *)

end (* foncteur Infixes *)
                              (* modules disponibles *)

module Big       : Int_type                (* bigints de Ocaml           *)
module Slong     : Int_type                (* longints assembleur        *)
module Clong     : Int_type                (* longints C                 *)
module Dlong     : Int_type                (* longints C double précision *)
```

```
module Gmp       : Int_type                    (* bigints de GMP              *)

                 (* chronométrage et sélection d'un module à l'exécution *)

external chrono : string → unit = "chrono"

module type Main_type = sig
  val main : string list → unit
end
module Start(Main : functor(E : Int_type) → Main_type) : sig
  val start : unit → unit
end
```

2.2 Précisions

Les modules `Big`, `Slong`, `Clong`, `Dlong` ou `Gmp` ne sont disponibles que sur les machines ayant une architecture matérielle permettant leur compilation et dans le cas de `Big` et `Gmp` si les bibliothèques `BigNum` et `GMP` sont installées. En règle générale les différents modules de type `Int_type` sont interchangeables c'est à dire qu'une opération donnée avec des arguments valides retourne un résultat sémantiquement indépendant du module utilisé et de l'architecture matérielle. Seules les opérations `gcd_ex`, `gcd_ex_in`, `cfrac`, `cfrac_in`, `look` et les fonctions de hachage, sérialisation et désérialisation échappent à cette règle (voir ci-après). Par contre les exceptions déclenchées en cas d'argument incorrect peuvent différer d'un module à l'autre.

Division

Les opérations de division `quo`, `quo_1`, `modulo`, `mod_1`, `quomod` et `quomod_1` et leurs équivalents sur place se conforment à la convention suivante :

`quomod` a b et `quomod_1` a b retournent le couple (q, r) tel que $a = bq + r$ avec $0 \leqslant |r| < |b|$, r ayant même signe que b. b ne doit pas être nul.

`quo` a b et `quo_1` a b retournent seulement q.

`modulo` a b et `mod_1` a b retournent seulement r.

Pgcd

Les opérations relatives au calcul du plus petit diviseur commun sont `gcd`, `gcd_ex`, `cfrac` et leurs équivalents sur place.

`cfrac` a b retourne un quintuplet (p, q, u, v, d) tel que :

1. d est le pgcd positif de a et b ;
2. $ua - vb = d$;
3. $pu - qv = 1$;
4. $pa = qb$ est le ppcm de a et b ayant même signe que ab.

Ces conditions assurent l'unicité de p, q, d lorsque a ou b n'est pas nul, mais les coefficients u et v ne sont pas uniques et peuvent différer d'un module à l'autre. Assurer leur unicité aurait été possible connaissant p

et q mais cela aurait conduit à effectuer des divisions supplémentaires potentiellement coûteuses en temps de calcul.

`gcd_ex` a b retourne le triplet (u, v, d).

`gcd` a b retourne d.

Comparaisons

Les fonctions `sgn`, `cmp` et `cmp_1` retournent exclusivement l'un des entiers 0, 1 ou -1 donc il est possible d'effectuer un filtrage sur le résultat de l'une de ces fonctions.

Les symboles de comparaison infixe (`<`, `<=`, `=`, `<>`, `>=` et `>`) ont un statut différent selon la version de Ocaml utilisée et selon que l'on a ouvert un module construit par le foncteur `Infixes` ou non.

- Pour les versions de Ocaml antérieures à 3.00, si l'on n'a pas ouvert de module `Infixes(E)`, les symboles `<`,...,`>` désignent les opérateurs de comparaison polymorphes traditionnels de Ocaml, de type `'a->'a->bool`. Ces opérateurs déclenchent une exception :

  ```
  Invalid_argument("equal: abstract value")
  ```

  s'ils sont appliqués à des grands entiers.

- Pour les versions de Ocaml antérieures à 3.00, si l'on a ouvert un module `Infixes(E)`, les symboles `<`,...,`>` désignent les opérateurs de comparaison entre grands entiers, de type `E.t->E.t`. Les opérateurs de comparaison polymorphes restent disponibles via les symboles `<!`,...,`>!`.

- A partir de Ocaml 3.00, les symboles `<`,...,`>` désignent les opérateurs de comparaison polymorphes traditionnels de Ocaml, que l'on ait ouvert un module `Infixes(E)` ou non, ils ne sont plus redéfinis par le foncteur `Infixes`. Les modules `Clong`, `Dlong`, `Slong` et `Gmp` comportent une interface avec la fonction de comparaison générique de Ocaml, et les opérateurs de comparaison polymorphes de Ocaml peuvent être utilisés avec les grands entiers de ces modules sans déclencher d'exception. De plus, les grands entiers de ces modules peuvent être stockés dans des structures ordonnées telles que les ensembles du module `Set` ou les tables d'association du module `Map` en utilisant la fonction de comparaison générique `compare`.

  Par contre la bibliothèque `Bignum` ne comporte pas d'interface avec la fonction de comparaison générique, et de ce fait les fonctions de comparaison polymorphes de Ocaml ne peuvent pas être utilisées avec des données de type `Big.t`. Si l'on souhaite pouvoir comparer de telles données avec les symboles infixes `<`,...,`>`, il faut redéfinir ces symboles soi-même, ou modifier la bibliothèque `Bignum` pour y inclure une interface à la fonction de comparaison générique de Ocaml. Une telle modification est décrite au chapitre **Installation**, section **Modification de la bibliothèque Bignum**, page 43.

Enfin, les symboles `<!`,...,`>!` restent liés aux opérateurs de comparaison polymorphes par le foncteur `Infixes`, pour préserver la compatibilité avec les versions antérieures de `Numerix`.

Représentation binaire

Les fonctions d'accès à la représentation binaire des grands entiers se conforment aux conventions suivantes :

`nbits` a retourne le nombre de bits significatifs de $|a|$; `nbits` $0 = 0$.

`lowbits` a et `highbits` a retournent les 31 bits de poids faible ou fort de $|a|$ si a est non nul et retournent 0 si $a = 0$. Pour `highbits`, lorsque $0 < |a| < 2^{30}$, des bits nuls sont ajoutés à droite pour atteindre un compte de 31 bits. Mathématiquement, pour a non nul et n entier tel que $2^{n-1} \leqslant |a| < 2^n$ on a :

$$\texttt{highbits}(a) = (|a| * 2^{\max(0,31-n)}) \text{ div } 2^{\max(n-31,0)}.$$

`nth_bit` a n et `nth_word` a n retournent le n-ème bit ou le n-ème mot de 16 bits de $|a|$, le bit ou mot de poids faible ayant l'indice zéro. Si n est négatif ou supérieur à la longueur de $|a|$ ces fonctions retournent 0.

`shl` a n retourne $a * 2^n$ si $n \geqslant 0$ et $\texttt{sgn}(a) * (|a| \text{ div } 2^{-n})$ si $n < 0$. `shr` a n est équivalent à `shl` a $(-n)$.

`split` a n découpe $|a|$ en deux parties et attache le signe de a à chacune de ces parties. n doit être positif ou nul.

`join` a b n retourne le nombre $a + b * 2^n$ y compris lorsque a et b ont des signes différents ou se recouvrent. n doit être positif ou nul.

Entiers pseudo-aléatoires

`random_bits` n retourne un entier pseudo-aléatoire compris entre 0 et $2^n - 1$ inclus, n doit être positif ou nul. Le générateur pseudo-aléatoire utilisé est celui de Ocaml, donc les valeurs retournées par `random_bits` sont identiques d'une exécution à l'autre d'un même programme ne modifiant pas la graine du générateur, et identiques pour tous les modules de grands entiers disponibles dans `Numerix`.

Entiers mutables

Les valeurs de type `tref` se comportent sémantiquement comme des références à des valeurs de type `t`, mais leur implémentation peut éventuellement être plus efficace et reste abstraite.

op`_in` r *args* calcule *op args* et range le résultat dans la mémoire référencée par r. Si ce résultat est plus grand que la capacité de cette mémoire un nouveau bloc mémoire est alloué pour contenir le résultat et r est mis à jour pour référencer ce nouveau bloc. Pour les opérations retournant des n-uplets : `quomod`, `gcd_ex`, `cfrac` et `split` l'équivalent sur place range les composantes du n-uplet dans les mémoires dont les références sont transmises en argument. Ces références doivent désigner des emplacements mémoire distincts.

`make_ref` v effectue une copie physique de v et retourne une référence à cette copie.

`copy_in` r v effectue une copie physique de v dans la mémoire référencée par r.

`copy_out` r retourne une copie physique de la mémoire référencée par r.

`look` r retourne la valeur référencée par r. Le résultat est volatile, c'est-à-dire que la valeur retournée n'est garantie valide que jusqu'à la prochaine opération sur place impliquant r. Les opérations de type lecture-modification-écriture, par exemple *op*`_in` r (`look` r) x, sont traitées correctement. Le foncteur `Infixes` donne deux abréviations pour l'opération `look` : `~` valide pour toutes les versions de Ocaml inférieures ou égales à 2.99 et `~~` valide pour toutes les versions de Ocaml inférieures ou égales à 3.00 (`~` est devenu un identificateur réservé dans Ocaml 3.00).

Chronométrage

`chrono` *msg* affiche sur le canal d'erreur standard le temps CPU en secondes depuis le début du processus, la différence avec le temps précédent et la chaîne *msg*. L'insertion de quelques appels à `chrono` dans un programme permet de connaître approximativement les temps d'exécution des différentes phases de ce programme.

Sélection d'un module à l'exécution

Le foncteur `Start` permet de sélectionner à l'exécution une implémentation particulière des grands entiers et d'exécuter le programme avec cette implémentation. L'argument de `Start` est un foncteur `Main` prenant en argument une implémentation des grands entiers et fournissant une implémentation de la fonction `main : string list` → `unit` qui constitue le point d'entrée du programme. L'argument de `main` est la liste des paramètres figurant sur la ligne de commande à l'exception de ceux désignant l'implémentation des grands entiers à utiliser. Le choix d'une implémentation particulière est effectué par la fonction `start` qui analyse la ligne de commande à la recherche d'une option `-e` *module* où *module* est le nom en minuscules d'un module grands entiers disponible, Si plusieurs options `-e` *module* figurent sur la ligne de commande seule la dernière est retenue ; si aucune option `-e` *module* n'est présente alors le premier module disponible est sélectionné. Si une option `-e` n'est pas suivie d'un nom de module reconnu alors un message d'erreur est affiché rappelant la liste des modules disponibles et le programme est arrêté.

Hachage, sérialisation et désérialisation

A partir de la version 3.00 de Ocaml les modules `Clong`, `Dlong`, `Slong` et `Gmp` comportent des interfaces avec la fonction de hachage générique et les fonctions de sérialisation et de désérialisation de Ocaml, donc les grands entiers de ces modules peuvent être inclus dans des tables de hachage utilisant la fonction `Hashtbl.hash`, et peuvent être exportés ou importés via les fonctions `output_value` et `input_value`. La bibliothèque `Bignum` comporte des interfaces avec les fonctions de sérialisation et de désérialisation de Ocaml, mais pas d'interface avec la fonction de hachage générique, donc les données de type

Big.t peuvent être exportées ou importées avec les fonctions output_value et input_value, mais elles ne peuvent pas être incluses dans des tables de hachage utilisant la fonction de hachage générique Hashtbl.hash. Il est possible de modifier la bibliothèque Bignum de façon à y incorporer une interface avec la fonction de hachage générique, voir le chapitre **Installation**, section **Modification de la bibliothèque Bignum**, page 43 à ce sujet.

En ce qui concerne la fonction de hachage générique, Hashtbl.hash, les clés de hachage retournées par cette fonction peuvent différer d'un module à l'autre pour des grands entiers sémantiquement égaux, du fait de représentations binaires différentes.

En ce qui concerne l'exportation et l'importation de grands entiers (fonctions output_value et input_value) et l'encodage des grands entiers par des suites d'octets (fonctions du module Marshal), le typage doit être préservé entre l'exportation ou l'encodage et l'importation ou le décodage, c'est-à-dire qu'il n'est pas possible de transformer un grand entier d'un module en grand entier d'un autre module avec ces fonctions.

2.3 Modules supplémentaires

Numerix comporte deux modules permettant de créer de nouveaux modules de type Int_type à partir de modules existant : Numcmp et Numcount.

2.3.1 Numcmp : comparaison entre deux modules

Le module Numcmp contient deux foncteurs :

```
module M(A:Int_type)(B:Int_type) : Int_type

module Start(Main : functor(E:Int_type) -> Main_type) : sig
  val start : unit -> unit
end
```

Numcmp.M prend en argument deux modules A et B de type Int_type et retourne un module C de même type dans lequel chaque opération *op* est réalisée par appels à $A.op$ et $B.op$ puis comparaison sémantique des résultats obtenus. Lorsqu'une comparaison échoue, c'est à dire lorsque $A.op$ et $B.op$ retournent des résultats sémantiquement différents pour des arguments supposés sémantiquement identiques, une exception est générée et les arguments et résultats dans chaque module sont affichés sur le canal d'erreur standard. Ce foncteur a été utilisé pour déboguer les modules en cours de développement par comparaison avec un module fiable, son usage en dehors de cette situation est déconseillé car la duplication des calculs et la comparaison des résultats consomment un temps important. Pour les opérations gcd_ex, gcd_ex_in, cfrac et cfrac_in les coefficients de Bézout ne sont pas comparés, ceux produits par B sont convertis en valeurs de type A.t pour former des résultats de type C.t.

Numcmp.Start est un foncteur analogue à Numerix.Start permettant de sélectionner à l'exécution par une option -e *module1_module2* quels sont les modules de grands entiers à comparer. Les couples de modules disponibles sont les suivants :

big_clong, big_dlong, big_slong, big_gmp
gmp_clong, gmp_dlong, gmp_slong.

Il n'y a pas de module par défaut. En cas d'option `-e` incorrecte le programme est arrêté et la liste des noms reconnus est affichée sur le canal d'erreur.

2.3.2 `Numcount` : comptage des opérations

Le module `Numcount` contient deux foncteurs :

```
module M(A:Int_type) : sig

  type statelt = {
    mutable n:float;  (* nombre d'appels   *)
    mutable s:float;  (* somme des tailles *)
    mutable m:int     (* taille maximale   *)
  }

  val cadd  : statelt (* add      sub                                *)
  val cmul  : statelt (* mul      sqr                                *)
  val cquo  : statelt (* quo      modulo       quomod                *)
  val cpow  : statelt (* pow      powmod       fact                  *)
  val croot : statelt (* sqrt     root                               *)
  val cgcd  : statelt (* gcd      gcd_ex       cfrac                 *)
  val cbin  : statelt (* shr      shl          split       join      *)
                      (* nbits    lowbits      highbits    nth_bit   *)
                      (* nth_word random_bits                        *)
  val cmisc : statelt (* abs      neg          make_ref    copy_in   *)
                      (* copy_out comparaisons conversions           *)

  val clear_stats : unit -> unit (* remise à zéro                         *)
  val print_stats : unit -> unit (* affiche les statistiques sur stderr *)

  module Int : Int_type (* module de grands entiers *)
end

module Start(Main : functor(E:Int_type) -> Main_type) : sig
  val start : unit -> unit
end
```

Le foncteur `Numcount.M` prend en argument un module A de type `Int_type` et retourne un module B`.Int` de même type dans lequel chaque opération *op* est réalisée par appel à $A.op$ et mise à jour de compteurs dépendant de l'opération effectuée. Le but de ce foncteur est de fournir des statistiques sur le nombre d'opérations entre grands entiers effectuées dans un programme. Ces opérations sont regroupées en huit catégories chacune associée à un compteur différent :

B`.cadd` compte les additions et soustractions ;

B.`cmul` compte les multiplications et les élévations au carré ;
B.`cquo` compte les divisions ;
B.`cpow` compte les exponentiations et les factorielles ;
B.`croot` compte les racines carrées et les racines p-èmes ;
B.`cgcd` compte les pgcd et les opérations associées ;
B.`cbin` compte les opérations sur les représentations binaires ;
B.`cmisc` compte toutes les autres opérations.

Chaque compteur B.`cxxx` comporte trois champs :

B.`cxxx.n` : nombres d'appels à l'une des fonctions associées à `cxxx` ;

B.`cxxx.s` : somme des tailles en bits des arguments ;

B.`cxxx.m` : maximum des tailles en bits des arguments.

Pour chaque appel à une fonction de B.`Int` le champ `n` du compteur associé est incrémenté d'une unité, le champ `s` est incrémenté de la moyenne des longueurs en bits des opérandes de type grand entier passés en paramètres et le champ `m` est mis à jour de façon à conserver le maximum des tailles en bits des opérandes des fonctions associées à ce compteur. Les opérandes de type `tref`, `int`, `string` ne sont pas pris en compte dans ces calculs de taille. Noter que les calculs effectués par B.`Int` sont intrinsèquement plus lents que les calculs correspondants effectués par A du fait de la gestion des compteurs et des appels indirects.

Il est possible de consulter et de modifier à tout moment chacun de ces compteurs afin de déterminer combien d'opérations dans chaque catégorie ont été effectuées depuis la dernière remise à zéro. La fonction B.`clear_stats` remet à zéro tous les compteurs et la fonction B.`print_stats` affiche sur le canal d'erreur les statistiques associées à chaque compteur (nombre d'opérations, taille moyenne et taille maximum d'un opérande).

`Numcount.Start` est un foncteur analogue à `Numerix.Start` permettant de sélectionner à l'exécution par une option `-e` *module* quel est le module E de grands entiers à utiliser avec les mêmes conventions que `Numerix.Start`. Si l'option `-count` figure sur la ligne de commande alors la fonction

`Main(Numcount.M(`E`).Int).main`

est appelée avec la liste des autres paramètres figurant sur la ligne de commande et la fonction `Numcount.M(`E`).print_stats` est appelée à la fin du programme. S'il n'y a pas d'option `-count` sur la ligne de commande alors la fonction `Main(`E`).main` est appelée comme avec `Numerix.Start`, dans ce cas il n'y a pas de statistiques affichées ni de ralentissement par rapport à l'utilisation de `Numerix.Start`.

Voici par exemple l'utilisation avec et sans l'option `-count` du programme de calcul de π décrit dans la section 6.6.4 :

```
exemples/pi/ocaml/pi-opt -e slong -count 1000000 -d -skip
    0.01     0.01 Count(Slong), surplace
    0.36     0.35 puiss 5
    3.79     3.43 sqrt
```

```
   26.48    22.69 série lb=6875847 ls=6875855
   31.93     5.45 quotient
   39.11     7.18 conversion
   op      count   avg.size   max.size
  add     424091        240    6875855
  mul     636133        323    6875847
  quo          3    2845643   10197784
  pow          3         13         19
 root          1    6643863    6643863
  gcd          0          -          -
  bin          1    4643867    4643867
 misc    1767125        353    6875855
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151

exemples/pi/ocaml/pi-opt -e slong 1000000 -d -skip
    0.00     0.00 Slong, surplace
    0.36     0.36 puiss 5
    3.83     3.47 sqrt
   23.96    20.13 série lb=6875847 ls=6875855
   29.41     5.45 quotient
   36.60     7.19 conversion
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151
```

Le calcul du premier million de décimales de π a donc nécessité 424091 additions/soustractions sur des opérandes de 240 bits en moyenne, 636133 multiplications/carrés sur des opérandes de 323 bits en moyenne, 3 divisions sur des opérandes de 2845643 en moyenne, etc. Le coût en temps dû à la tenue de ces statistiques est d'environ 2.5 secondes.

2.4 Compilation

Les programmes Ocaml utilisant `Numerix` doivent être compilés avec les commandes suivantes :

```
ocamlc -custom options nums.cma numerix.cma fichiers source \
       -cclib -lmlnumx -cclib -lnums -cclib -lgmp

ocamlopt options nums.cmxa numerix.cmxa fichiers source \
       -cclib -lmlnumx -cclib -lnums -cclib -lgmp
```

Les fichiers `nums.cma`, `nums.cmxa`, `numerix.cma` et `numerix.cmxa` contiennent sous forme compilée les parties Ocaml des bibliothèques `BigNum` (bibliothèque fournie avec Ocaml) et `Numerix`. Il peut être nécessaire d'indiquer aux compilateurs où se trouvent les fichiers `numerix.cma` et `numerix.cmxa` au moyen d'une option `-I` *chemin*.

Les options `-cclib -lmlnumx`, `-cclib -lnums` et `-cclib -lgmp` demandent à l'éditeur de liens de rechercher dans les fichiers `libmlnumx.a`, `libnums.a` et `libgmp.a` les primitives C dont il aurait besoin. `libnums.a` contient sous forme compilée la partie C de `BigNum`, `libmlnumx.a` la partie C de `Numerix` et `libgmp.a` la partie C de `GMP`. Il peut être nécessaire d'indiquer à l'éditeur de liens dans quels répertoires chercher ces fichiers `libxxx.a` au moyen d'options `-ccopt -L`*chemin*. Si `GMP` n'est pas installé ou si son interface avec Ocaml n'est pas incluse dans votre version de `Numerix`, alors l'option `-cclib -lgmp` doit être omise. De même, les paramètres `nums.cma`, `nums.cmxa` et `-cclib -lnums` doivent être omis si `BigNum` n'est pas installé.

A partir de la version 3.00 de Ocaml, les options `-custom`, `-cclib -lxxx` et `-ccopt xxx` sont enregistrées dans les fichiers `numerix.cma` et `numerix.cmxa` au moment de la compilation de `Numerix` donc n'ont pas à être redonnées pour compiler un programme utilisant `Numerix`.

2.5 Exemple

```
(* +-----------------------------------------------------------------------+
   |  Fichier simple.ml: Démonstration de la bibliothèque Numerix avec Ocaml  |
   +-----------------------------------------------------------------------+ *)

(* calcul de (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales *)

open Numerix

module Main(E:Int_type) = struct
  module I = Infixes(E)
  open E
  open I

  (* version fonctionnelle *)
  let calc_func n =
    let dix_n  = (of_int(5) ** n) << n in   (* 10^n  *)
    let dix_2n = sqr(dix_n)            in   (* 10^2n *)
    let sqrt_2 = sqrt(dix_2n *. 200)   in   (* sqrt(2) avec n+1 décimales *)
    let sqrt_3 = sqrt(dix_2n *. 300)   in   (* sqrt(3) avec n+1 décimales *)
    (dix_n * (sqrt_3 + sqrt_2)) / (sqrt_3 - sqrt_2)

  (* version sur place *)
  let calc_in n =
    let dix_n  = make_ref(of_int 0) in
    let dix_2n = make_ref(of_int 0) in
    let sqrt_2 = make_ref(of_int 0) in
    let sqrt_3 = make_ref(of_int 0) in
    let x      = make_ref(of_int 0) in
```

```
    let y      = make_ref(of_int 0) in

    pow_in   dix_n    (of_int 5)  n;         (* dix_n <- 5^n ...       *)
    shl_in   dix_n    ~~dix_n     n;         (*        ... * 2^n       *)
    sqr_in   dix_2n   ~~dix_n;               (* dix_2n <- 10^(2n)      *)
    mul_1_in sqrt_2   ~~dix_2n    200;       (* sqrt_2 <- ...          *)
    sqrt_in  sqrt_2   ~~sqrt_2;              (* .. sqrt(2*10^(2n+2))  *)
    mul_1_in sqrt_3   ~~dix_2n    300;       (* sqrt_3 <- ...          *)
    sqrt_in  sqrt_3   ~~sqrt_3;              (* .. sqrt(3*10^(2n+2))  *)
    add_in   x        ~~sqrt_3    ~~sqrt_2;  (* x <- sqrt_2 + sqrt_2   *)
    sub_in   y        ~~sqrt_3    ~~sqrt_2;  (* y <- sqrt_3 - sqrt_2   *)
    mul_in   x        ~~dix_n     ~~x;       (* x <- trunc(10^n*x ... *)
    quo_in   x        ~~x         ~~y;       (*              ... /y)  *)
    copy_out x                               (* résultat               *)

  (* fonction principale *)
  let main arglist =

    let rec parse (n,f) = function
      | []          -> (n,  f)
      | "-test"::s  -> parse (30, calc_func) s
      | "-func"::s  -> parse (n,  calc_func) s
      | "-in"  ::s  -> parse (n,  calc_in)   s
      | "-n"::x::s  -> parse (int_of_string x, f) s
      | _ -> prerr_endline "syntaxe : simple [-e entiers] [-test] [-func|-in] [-n <n>]";
             exit(1)
    in
    let (n,f) = parse (30,calc_func) arglist in
    print_endline(string_of(f n))

end (* Main *)

module S = Numcount.Start(Main)
let _ = S.start()
```

Compilation et exécution (avec Ocaml-3.00) :

```
> ocamlc -o simple-byte nums.cma numerix.cma simple.ml
> ./simple-byte -e slong
9898979485566356196394568149410
>
> ocamlopt -o simple-opt nums.cmxa numerix.cmxa simple.ml
> ./simple-opt -e gmp -n 50 -count
   op      count   avg.size   max.size
  add          2        170        171
  mul          4        250        333
  quo          1        253        338
  pow          1          3          3
 root          2        340        341
```

```
  gcd          0          -           -
  bin          1        117         117
 misc          2         85         170
989897948556635619639456814941178278393189496131335
```

2.6 Système interactif

`ocamlnumx` est une version spécialisée du système interactif `ocaml` lié avec le fichier objet `numerix.cma` et les bibliothèques compilées `libnums.a`, `libgmp.a` et `libmlnumx.a`. Il permet d'utiliser tous les modules de `Numerix`, le choix d'une implémentation des grands entiers se faisant par une directive `open` appropriée.

```
> ocamlnumx
ocamlnumx : Objective Caml with big integer libraries
Numerix submodules : Big Clong Dlong Slong Gmp
Numerix version    : 0.19 (24 Aout 2001)

        Objective Caml version 3.02

# open Numerix;;
# open Big;;
# fact(30);;
- : Numerix.Big.t = 265252859812191058636308480000000
# module I = Infixes(Big) open I;;
module I :
  sig
    val ( + ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    val ( - ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    val ( * ) : Numerix.Big.t -> Numerix.Big.t -> Numerix.Big.t
    ...
    val ( ~~ ) : Numerix.Big.tref -> Numerix.Big.t
  end
# (of_int 1) << 100;;
- : Numerix.Big.t = 1267650600228229401496703205376
# #quit;;
```

Chapitre 3

Utilisation avec Camllight

L'interface pour Camllight de `Numerix` est dérivée de celle pour Ocaml en retirant ou en adaptant les fonctionnalités spécifiques au langage Ocaml. On se reportera au chapitre précédent pour la liste des fonctions disponibles, seules les différences avec la version Ocaml sont présentées ici. Cette interface a été testée avec succès avec les versions 0.73, 0.74 et 0.75 de Camllight.

3.1 Modules

Camllight dispose d'un système limité de modules et en particulier ne supporte pas les notions de sous-module ni de module paramétré. Il reste toutefois possible d'écrire du code indépendant d'une implémentation particulière des grands entiers en utilisant les noms courts des fonctions, les noms longs étant déduits par le compilateur en fonction de directives `#open` figurant dans le fichier source. Il suffit de modifier ces directives (éventuellement de façon automatique au moyen d'un préprocesseur) et de recompiler le code source pour changer l'implémentation des grands entiers utilisée.

Les modules disponibles portent les mêmes noms qu'en Ocaml sans majuscule : `big`, `clong`, `dlong`, `slong` et `gmp`. Il n'y a pas d'équivalents aux modules construits en Ocaml avec les foncteurs `Numcmp.M` et `Numcount.M`. Les notations infixes sont accessibles en ouvrant le module `infxxx` où `xxx` est le nom du module implémentant les grands entiers.

3.2 Fonctions

Les fonctions décrites dans la signature `Int_type` en Ocaml ont été conservées en Camllight avec seulement deux différences :

La division sans reste est notée `quo` en Ocaml et `div` en Camllight. La raison de cette différence est que l'identificateur `quo` a un statut infixe en Camllight. Les autres noms dérivés de `quo` : `quomod`, `quo_1`, `quomod_1`, `quo_in`, `quomod_in` et `quomod_1_in` ont été conservés.

La consultation d'une référence est notée `look` ou `~` en Ocaml, mais `look` ou `?` en Camllight. Il y a deux raisons à cette différence : l'identificateur `?` est réservé et donc indisponible en Ocaml et l'identificateur ~ a un statut préfixe en Ocaml mais infixe en Camllight.

3.3 Compilation

Les programmes Caml utilisant `Numerix` doivent être compilés avec la commande suivante :

`camlc -custom` *options* `nums.zo numerix.zo` *fichiers source* `-lclnumx -lnums -lgmp`

Les fichiers `nums.zo` et `numerix.zo` contiennent sous forme compilée les parties Caml des bibliothèques `BigNum` (bibliothèque fournie avec Camllight) et `Numerix`. Il peut être nécessaire d'indiquer au compilateur où se trouve le fichier `numerix.zo` au moyen d'une option `-I` *chemin*.

Les options `-lclnumx`, `-lnums` et `-lgmp` demandent à l'éditeur de liens de rechercher dans les fichiers `libclnumx.a`, `libnums.a` et `libgmp.a` les primitives C dont il aurait besoin. `libnums.a` contient sous forme compilée la partie C de `BigNum`, `libclnumx.a` la partie C de `Numerix` et `libgmp.a` la partie C de `GMP`. Il peut être nécessaire d'indiquer à l'éditeur de liens dans quels répertoires chercher ces fichiers `libxxx.a` au moyen d'options `-ccopt -L`*chemin*. Si `GMP` n'est pas installé ou si son interface avec Camllight n'est pas incluse dans votre version de `Numerix`, alors l'option `-lgmp` doit être omise. De même, les paramètres `nums.zo` et `-cclib -lnums` doivent être omis si `BigNum` n'est pas installé.

3.4 Exemple

```
(* +-------------------------------------------------------------------------+
   |  Fichier simple.ml: Démonstration de la bibliothèque Numerix avec Camllight  |
   +-------------------------------------------------------------------------+ *)

(* calcul de (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales *)

#open "slong";;
#open "infslong";;

  (* version fonctionnelle *)
  let calc_func n =
    let dix_n  = (of_int(5) ** n) << n in   (* 10^n  *)
    let dix_2n = sqr(dix_n)            in   (* 10^2n *)
    let sqrt_2 = sqrt(dix_2n *. 200)   in   (* sqrt(2) avec n+1 décimales *)
    let sqrt_3 = sqrt(dix_2n *. 300)   in   (* sqrt(3) avec n+1 décimales *)
    (dix_n * (sqrt_3 + sqrt_2)) / (sqrt_3 - sqrt_2);;

  (* version sur place *)
  let calc_in n =
    let dix_n  = make_ref(of_int 0) in
    let dix_2n = make_ref(of_int 0) in
```

```
    let sqrt_2 = make_ref(of_int 0) in
    let sqrt_3 = make_ref(of_int 0) in
    let x      = make_ref(of_int 0) in
    let y      = make_ref(of_int 0) in

    pow_in   dix_n    (of_int 5)  n;        (* dix_n <- 5^n ...       *)
    shl_in   dix_n    ?dix_n      n;        (*        ... * 2^n       *)
    sqr_in   dix_2n   ?dix_n;               (* dix_2n <- 10^(2n)      *)
    mul_1_in sqrt_2   ?dix_2n     200;      (* sqrt_2 <- ...          *)
    sqrt_in  sqrt_2   ?sqrt_2;              (* .. sqrt(2*10^(2n+2))   *)
    mul_1_in sqrt_3   ?dix_2n     300;      (* sqrt_3 <- ...          *)
    sqrt_in  sqrt_3   ?sqrt_3;              (* .. sqrt(3*10^(2n+2))   *)
    add_in   x        ?sqrt_3     ?sqrt_2;  (* x <- sqrt_3 + sqrt_2   *)
    sub_in   y        ?sqrt_3     ?sqrt_2;  (* y <- sqrt_3 - sqrt_2   *)
    mul_in   x        ?dix_n      ?x;       (* x <- trunc(10^n*x ...  *)
    quo_in   x        ?x          ?y;       (*               ... /y)  *)
    copy_out x;;                            (* résultat               *)

  (* fonction principale *)
  let main arglist =

    let rec parse (n,f) = function
      | []          -> (n,  f)
      | "-test"::s  -> parse (30, calc_func) s
      | "-func"::s  -> parse (n,  calc_func) s
      | "-in"  ::s  -> parse (n,  calc_in)   s
      | "-n"::x::s  -> parse (int_of_string x, f) s
      | _ -> prerr_endline "syntaxe : simple [-test] [-func|-in] [-n <n>]";
             exit(1)
    in
    let (n,f) = parse (30,calc_func) arglist in
    print_endline(string_of(f n));
    flush stdout;;

main (tl(list_of_vect sys__command_line));;
```

Compilation et exécution :

```
> camlc -custom -o simple nums.zo numerix.zo simple.ml \
        -lclnumx -lnums -lgmp
> ./simple
989897948556635619639456814941O
```

Noter que les trois bibliothèques `libclnumx.a`, `libnums.a` et `libgmp.a` doivent être fournies à l'éditeur de liens même si seul le module `slong` est utilisé, car les autres modules sont présents dans `numerix.zo` et font référence à des fonctions de ces trois bibliothèques.

3.5 Système interactif

De même que pour la version Ocaml, un système interactif spécialisé est disponible :

```
> camllight ~/lib/camlnumx
>       Caml Light version 0.74

camlnumx : Caml toplevel with big integer libraries
Numerix submodules : big clong dlong slong gmp
Numerix version    : 0.19 (24 Aout 2001)

##open "big";;
##open "infbig";;
#fact(30);;
- : t = 265252859812191058636308480000000
#(of_int 1) << 100;;
- : t = 1267650600228229401496703205376
#quit();;
```

Chapitre 4

Utilisation en C

L'interface C de Numerix a été dérivée de l'interface Ocaml en ajoutant un gestionnaire mémoire simplifié destiné à palier l'absence de celui de Ocaml et en se limitant aux opérations sur place. La raison d'être de cette interface est de permettre une comparaison non biaisée entre Numerix et GMP dont l'environnement naturel d'utilisation est le langage C, et de pouvoir compiler et exécuter des programmes de test sur une machine ne disposant pas de Ocaml.

4.1 Interface

Les trois modules `clong`, `dlong` et `slong` sont disponibles, pour autant que le compilateur C utilisé et l'architecture matérielle de la machine cible le permettent. Le choix du module à utiliser et la taille des mots machine doivent être indiqués à la compilation au moyen des directives `#define` suivantes :

`#define bits_32` et `#define bits_64` indiquent la taille du mot machine. Une et une seule de ces directives doit être spécifiée.

`#define use_clong`, `#define use_dlong` et `#define use_slong` sélectionnent le module implémentant les grands entiers. Au plus une de ces directives doit être spécifiée, le module sélectionné en l'absence de spécification est `Clong`.

Ces directives peuvent être incluses en tête de chaque fichier source ou transmises au préprocesseur à l'aide d'options `-Dxxx` sur la ligne de commande de compilation.

Le fichier `c-long_int.h` définit le type de données `xint` représentant un grand entier et donne les prototypes des fonctions opérant sur les grands entiers. Les noms des fonctions sont préfixés par une chaîne de trois caractères identifiant le module `Clong`, `Dlong` ou `Slong` auquel elles appartiennent et le niveau d'abstraction auquel elles opèrent :

`cn_nnn` : fonction `nnn` du module `Clong` opérant sur les entiers naturels (tableaux de chiffres) ;

`cz_nnn` : fonction `nnn` du module `Clong` opérant sur les entiers relatifs (tableaux de chiffres avec longueur et signe accolés) ;

`cx_nnn` : fonction `nnn` du module `Clong` opérant sur les grands entiers (entiers relatifs extensibles et relogeables).

`dn_nnn`, `dz_nnn`, `dx_nnn`, idem pour le module `Dlong`.

`sn_nnn`, `sz_nnn`, `sx_nnn`, idem pour le module `Slong`.

Normalement le programmeur ne devrait avoir affaire qu'aux grands entiers, de type `xint`, et n'utiliser que les fonctions `cx_nnn` ou `dx_nnn` ou `sx_nnn` qui opèrent sur ces grands entiers. Les fonctions de niveau `n` ou `z` sont à usage interne uniquement et ne sont pas documentées ici. De façon à permettre l'écriture d'un code indépendant de l'implémentation des grands entiers le fichier `c-long_int.h` définit une macro `xx` dont le code d'expansion est `cx_` ou `dx_` or `sx_` selon que l'un des symboles `use_clong`, `use_dlong` ou `use_slong` est défini. On écrira donc :

`xx(add)(&x,a,b) ;`

pour additionner `a` et `b` dans `x`, ce code étant transformé par le préprocesseur en :

`cx_add(&x,a,b) ;` ou `dx_add(&x,a,b) ;` ou `sx_add(&x,a,b) ;`

Il conseillé d'utiliser systématiquement la macro `xx` et de ne pas coder directement les identificateurs expansés, cela permet de recompiler son programme avec une autre implémentation des grands entiers par simple modification de la directive `#define use_xxx`. En tout état de cause les fonctions d'un module ne peuvent opérer sur les données d'un autre module et il n'y a pas de mécanisme permettant de différencier le type de données `xint` selon le module.

Voici un extrait de `c-long_int.h` donnant les prototypes des fonctions publiques :

```
/* préfixe des fonctions publiques  */
#if defined(use_slong)
#define xx(nom) sx_##nom
#elif defined(use_dlong)
#define xx(nom) dx_##nom
#else
#define xx(nom) cx_##nom
#endif

/* entier extensible  */
typedef struct {...} * xint;

/* allocation/désallocation */
xint xx(new)(void);
void xx(free)(xint *x);

/* initialisation à partir d'un long, d'un entier ou d'une chaine */
xint xx(of_int)     (long a);
xint xx(of_xint)    (xint a);
xint xx(of_string)  (char *s);
```

```
/* modification d'un entier déjà initialisé */
void xx(copy)       (xint *x, xint a);             /* x ← a         */
void xx(copy_int)   (xint *x, long a);             /* x ← a         */
void xx(copy_string)(xint *x, char *s);            /* x ← val(s)    */

/* opérations usuelles */
void xx(add)   (xint *x, xint a, xint b);          /* x ← a+b       */
void xx(sub)   (xint *x, xint a, xint b);          /* x ← a-b       */
void xx(mul)   (xint *x, xint a, xint b);          /* x ← ab        */
void xx(quo)   (xint *x, xint a, xint b);          /* x ← a div b   */
void xx(quomod)(xint *x, xint *y,xint a, xint b);  /* x ← a div b, y ← a mod b */
void xx(gcd)   (xint *z, xint a, xint b);          /* x ← pgcd(a,b) */
void xx(powmod)(xint *x, xint a, xint b, xint c);  /* x ← a^b mod c */

/* pgcd étendu : z ← pgcd(a,b), xa-by = z, xr-ys = 1 */
void xx(gcd_ex)(xint *x, xint *y, xint *z, xint a, xint b);
void xx(cfrac) (xint *r, xint *s, xint *x, xint *y, xint *z, xint a, xint b);

/* opérations unaires */
void xx(abs)   (xint *x, xint a);                  /* x ← |a|       */
void xx(neg)   (xint *x, xint a);                  /* x ← -a        */
void xx(sqr)   (xint *x, xint a);                  /* x ← a^2       */
void xx(sqrt)  (xint *x, xint a);                  /* x ← trunc(a^1/2) */

/* opérations mixtes */
void xx(add_1) (xint *x, xint a, long b);          /* x ← a+b       */
void xx(sub_1) (xint *x, xint a, long b);          /* x ← a-b       */
void xx(mul_1) (xint *x, xint a, long b);          /* x ← ab        */
long xx(quomod_1) (xint *x, xint a, long b);       /* x ← a div b, retourne a mod b */
void xx(shl)   (xint *x, xint a, long b);          /* x ← a*2^b     */
void xx(shr)   (xint *x, xint a, long b);          /* x ← a div 2^b */
void xx(split) (xint *x, xint *y,xint a, long b);  /* x ← a mod 2^b, y ← a div 2^b */
void xx(join)  (xint *x, xint a, xint b, long c);  /* x ← a + b*2^c */
void xx(pow)   (xint *x, xint a, long b);          /* x ← a^b       */
void xx(root)  (xint *x, xint a, long b);          /* x ← trunc(a^1/b) */
void xx(fact)  (xint *x, long a);                  /* x ← a!        */

/* comparaison */
int xx(eq)     (xint a, xint b);                   /* a = b ?        */
int xx(neq)    (xint a, xint b);                   /* a ≠ b ?        */
int xx(inf)    (xint a, xint b);                   /* a < b ?        */
int xx(infeq)  (xint a, xint b);                   /* a ⩽ b ?        */
int xx(sup)    (xint a, xint b);                   /* a > b ?        */
int xx(supeq)  (xint a, xint b);                   /* a ⩾ b ?        */
int xx(cmp)    (xint a, xint b);                   /* signe(a-b)      */
int xx(sgn)    (xint a);                           /* signe(a)        */

int xx(eq_1)   (xint a, long b);                   /* a = b ?        */
int xx(neq_1)  (xint a, long b);                   /* a ≠ b ?        */
```

```
int xx(inf_1)  (xint a, long b);                        /* a < b ?        */
int xx(infeq_1)(xint a, long b);                        /* a ⩽ b ?        */
int xx(sup_1)  (xint a, long b);                        /* a > b ?        */
int xx(supeq_1)(xint a, long b);                        /* a ⩾ b ?        */
int xx(cmp_1)(xint a, long b);                          /* sign(a-b)      */

/* accès à la structure binaire, lowbits, highbits et int_of retournent
  des mots de 31 bits seulement pour compatibilité avec Ocaml */
long xx(nbits)   (xint a);                              /* taille de a    */
long xx(lowbits) (xint a);                              /* voir Ocaml     */
long xx(highbits)(xint a);                              /* voir Ocaml     */
long xx(nth_word)(xint a, long n);                      /* bits 16b..16b+15*/
long xx(int_of)  (xint a);                              /* bits 0..30     */

/* conversion entier -> chaine, désallouer la chaine par free_string */
char *xx(string_of)(xint a);
void xx(free_string)(char *s);

/* chronométrage */
void chrono(char *msg);
```

4.2 Précisions

Une variable de type `xint` est un pointeur sur une structure de données gérée par le gestionnaire mémoire intégré à la version C de `Numerix`. L'initialisation d'une variable `x` de type `xint` s'effectue normalement en deux étapes :

1. initialisation du pointeur `x` par appel à `xx(new)()` ;
2. affectation d'une valeur en indiquant l'adresse de `x`, `&x`, en paramètre résultat d'un calcul, par exemple `xx(add)(&x,a,b)`.

Les fonctions `xx(of_int)`, `xx(of_xint)` et `xx(of_string)` réalisent à la fois l'initialisation du pointeur et l'affectation d'une valeur à partir d'un entier ordinaire, d'un autre grand entier ou d'une chaîne de caractères. L'instruction :

```
xint x = xx(of_int)(i) ;
```

est formellement équivalente aux instructions :

```
xint x = xx(new)() ; xx(copy_int)(&x,i) ;
```

et de même pour `xx(of_xint)` et `xx(of_string)`. L'exécution d'une opération par un appel tel que :

```
xx(quo)(&x,a,b) ;
```

est valide si et seulement si les conditions suivantes sont remplies :

1. les variables `a` et `b` sont initialisées et désignent des valeurs valides pour l'opération en question (`a` désigne une valeur quelconque et `b` une valeur quelconque non nulle) ;

2. la variable `x` contient un pointeur initialisé.

Il n'est pas nécessaire que `x` ait reçu une valeur préalablement à cette opération. Si c'est le cas alors le bloc mémoire contenant cette valeur est surchargé avec le résultat de l'opération s'il est suffisamment grand, sinon un nouveau bloc mémoire est alloué pour recevoir le résultat, le pointeur `x` est modifié pour désigner ce nouveau bloc et l'ancien bloc est désalloué.

Les opérations de type lecture-modification-écriture où une même variable figure à la fois en positions opérande et résultat sont correctement traitées. Par contre pour les opérations produisant plusieurs résultats (`quomod`, `gcd_ex`, `cfrac` et `split`) une même variable ne peut figurer plusieurs fois en position résultat. Par exemple l'instruction suivante est illégale :

```
xx(quomod)(&x,&x,a,b); /* illégal */
```

La fonction `xx(free)` permet de forcer la libération du bloc mémoire occupé par une valeur qui n'est plus utile. L'instruction :

```
xx(free)(&x);
```

a pour effet de désallouer la valeur désignée par `x` s'il y en a une et de réinitialiser le pointeur `x`. La variable `x` reste utilisable après cette instruction pour recevoir une nouvelle valeur.

La conversion d'un grand entier en chaîne de caractères est réalisée par la fonction `xx(string_of)` qui alloue une chaîne de caractères de taille convenable et y copie la représentation décimale de son argument. Cette chaîne devra être libérée après usage par appel à la fonction `xx(free_string)`.

Enfin, la fonction `chrono` affiche sur le canal d'erreur standard le temps CPU en secondes écoulé depuis le début du processus, la différence avec le temps précédent et le message passé en argument.

4.3 Compilation

Les programmes C utilisant `Numerix` doivent être compilés avec la commande suivante :

`gcc` *options* `-Dbits_xx -Duse_yyy` *fichiers source* `-lcnumx`

`-Dbits_xx` indique la taille d'un mot machine, `xx` = 32 ou 64. `-Duse_yyy` indique le module à utiliser, `yyy` = `clong` ou `dlong` ou `slong`. `-lcnumx` indique à l'éditeur de liens de rechercher dans le fichier `libcnumx.a` les fonctions compilées dont il aurait besoin. Il peut être nécessaire d'indiquer à l'éditeur de liens dans quel répertoire chercher ce fichier `libcnumx.a` au moyen d'une option `-L`*chemin*. De même il peut être nécessaire d'indiquer au préprocesseur au moyen d'une option `-I`*chemin* où trouver le fichier d'en-tête `c-long_int.h`.

4.4 Exemple

```
 /* +--------------------------------------------------------------------+
    |  Fichier simple.c: Démonstration de la bibliothèque Numerix avec C  |
    +--------------------------------------------------------------------+ */

/* calcul de (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales */

#include <stdlib.h>
#include <stdio.h>
#include "c-long_int.h"

void calcul(xint *res, long n) {
   xint dix_n  = xx(new)();
   xint dix_2n = xx(new)();
   xint sqrt_2 = xx(new)();
   xint sqrt_3 = xx(new)();
   xint x      = xx(new)();
   xint y      = xx(new)();

   xx(copy_int)(&dix_n,5);
   xx(pow)  (&dix_n,  dix_n,  n);            /* dix_n <- 5^n ...       */
   xx(shl)  (&dix_n,  dix_n,  n);            /*        ... * 2^n       */
   xx(sqr)  (&dix_2n, dix_n);                /* dix_2n <- 10^(2n)      */
   xx(mul_1)(&sqrt_2, dix_2n, 200);          /* sqrt_2 <- ...          */
   xx(sqrt) (&sqrt_2, sqrt_2);               /* .. sqrt(2*10^(2n+2))   */
   xx(mul_1)(&sqrt_3, dix_2n, 300);          /* sqrt_3 <- ...          */
   xx(sqrt) (&sqrt_3, sqrt_3);               /* .. sqrt(3*10^(2n+2))   */
   xx(add)  (&x,      sqrt_3, sqrt_2);       /* x <- sqrt_3 + sqrt_2   */
   xx(sub)  (&y,      sqrt_3, sqrt_2);       /* y <- sqrt_3 - sqrt_2   */
   xx(mul)  (&x,      dix_n,  x);            /* res <- trunc(10^n*x...*/
   xx(quo)  (res,     x,      y);            /*              ... /y) */

   xx(free)(&dix_n);
   xx(free)(&dix_2n);
   xx(free)(&sqrt_2);
   xx(free)(&sqrt_3);
   xx(free)(&x);
   xx(free)(&y);
}


int main(int argc, char **argv) {
  long n = 30;
  xint x = xx(new)();
  char *s;

  for(argc--, argv++; (argc); argc--, argv++) {
    if      (strcmp(*argv,"-test") == 0) {n = 30;}
    else if ((strcmp(*argv,"-n") == 0) && (argc > 0)) {
```

```
      argc--, argv++;
      n = atol(*argv);
    } else {
      fprintf(stderr,"syntaxe : simple [-test] [-n <n>]\n");
      fflush(stderr);
      exit(1);
    }
  }

  calcul(&x,n);
  s = xx(string_of)(x);
  fprintf(stdout,"%s\n",s);
  fflush(stdout);

  xx(free)(&x);
  xx(free_string)(s);
  return(0);
}
```

Compilation et exécution :

```
> gcc -O2 -Wall -Dbits_32 -Duse_slong -o simple simple.c -lcnumx
> ./simple
989897948556635619639456814941O
```

Chapitre 5

Utilisation en Pascal

L'interface Pascal de `Numerix` est construite à partir de l'interface C et fournit les mêmes fonctionnalités que cette dernière. Elle a été développée sur un PC-Linux (processeur Athlon) avec le compilateur Free Pascal version 1.0.2.

5.1 Interface

Trois unités sont définies : `clong`, `dlong` et `slong`. Chaque unité contient la déclaration du type grand entier correspondant et les déclarations des procédures et fonctions associées. Les identificateurs exportés sont les mêmes d'une unité à l'autre, ce qui permet d'écrire des programmes utilisateurs indépendants de l'unité choisie, seule la clause `uses` devant être modifiée pour changer l'implémentation des grands entiers.

```
unit clong; (* ou dlong ou slong *)
interface

(* entier extensible  *)
type xint = ...;

(* allocation/désallocation *)
function  xnew : xint;
procedure xfree(var x : xint);

(* initialisation à partir d'un entier, d'un long ou d'une chaine *)
function  of_int     (a : longint) : xint;
function  of_xint    (a : xint   ) : xint;
function  of_string  (s : pchar  ) : xint;

(* modification d'un entier déjà initialisé *)
procedure copy       (var x : xint; a : xint);            (* x ← a         *)
procedure copy_int   (var x : xint; a : longint);         (* x ← a         *)
procedure copy_string(var x : xint; s : pchar);           (* x ← val(s)    *)

(* opérations usuelles *)
```

```
procedure add     (var x   : xint; a,b   : xint);              (* x ← a+b       *)
procedure sub     (var x   : xint; a,b   : xint);              (* x ← a-b       *)
procedure mul     (var x   : xint; a,b   : xint);              (* x ← ab        *)
procedure quo     (var x   : xint; a,b   : xint);              (* x ← a div b   *)
procedure quomod  (var x,y : xint; a,b   : xint);              (* x ← a div b,  *)
                                                               (* y ← a mod b   *)
procedure gcd     (var z   : xint; a,b   : xint);              (* x ← pgcd(a,b) *)
procedure powmod  (var x   : xint; a,b,c : xint);              (* x ← a^b mod c *)

(* pgcd étendu : z ← pgcd(a,b), xa-by = z, xr-ys = 1 *)
procedure gcd_ex  (var x,y,z     : xint; a,b : xint);
procedure cfrac   (var r,s,x,y,z : xint; a,b : xint);

(* opérations unaires *)
procedure abs     (var x   : xint; a     : xint);              (* x ← |a|       *)
procedure neg     (var x   : xint; a     : xint);              (* x ← -a        *)
procedure sqr     (var x   : xint; a     : xint);              (* x ← a^2       *)
procedure sqrt    (var x   : xint; a     : xint);              (* x ← trunc(a^1/2) *)

(* opérations mixtes *)
procedure add_1   (var x   : xint; a     : xint; b : longint); (* x ← a+b       *)
procedure sub_1   (var x   : xint; a     : xint; b : longint); (* x ← a-b       *)
procedure mul_1   (var x   : xint; a     : xint; b : longint); (* x ← ab        *)
function  quomod_1(var x   : xint; a     : xint; b : longint) : longint;
                                                               (* x ← a div b,  *)
                                                               (* retourne a mod b *)
procedure shiftl  (var x   : xint; a     : xint; b : longint); (* x ← a*2^b     *)
procedure shiftr  (var x   : xint; a     : xint; b : longint); (* x ← a div 2^b *)
procedure split   (var x,y : xint; a     : xint; b : longint); (* x ← a mod 2^b,*)
                                                               (* y ← a div 2^b *)
procedure join    (var x   : xint; a,b   : xint; c : longint); (* x ← a + b*2^c *)
procedure pow     (var x   : xint; a     : xint; b : longint); (* x ← a^b       *)
procedure root    (var x   : xint; a     : xint; b : longint); (* x ← trunc(a^1/b) *)
procedure fact    (var x   : xint; a     : xint);              (* x ← a!        *)

(* comparaison *)
function eq   (a,b : xint) : boolean;                          (* a = b ?        *)
function neq  (a,b : xint) : boolean;                          (* a ≠ b ?        *)
function inf  (a,b : xint) : boolean;                          (* a < b ?        *)
function infeq(a,b : xint) : boolean;                          (* a ⩽ b ?        *)
function sup  (a,b : xint) : boolean;                          (* a > b ?        *)
function supeq(a,b : xint) : boolean;                          (* a ⩾ b ?        *)
function cmp  (a,b : xint) : longint;                          (* signe(a-b)      *)
function sgn  (a   : xint) : longint;                          (* signe(a)        *)

function eq_1   (a : xint; b : longint) : boolean;             (* a = b ?        *)
function neq_1  (a : xint; b : longint) : boolean;             (* a ≠ b ?        *)
function inf_1  (a : xint; b : longint) : boolean;             (* a < b ?        *)
function infeq_1(a : xint; b : longint) : boolean;             (* a ⩽ b ?        *)
function sup_1  (a : xint; b : longint) : boolean;             (* a > b ?        *)
```

```
function supeq_1(a : xint; b : longint) : boolean;            (* a ⩾ b ?        *)
function cmp_1  (a : xint; b : longint) : longint;            (* signe(a-b)     *)

(* accès à la structure binaire, lowbits, highbits et int_of retournent
  des mots de 31 bits seulement pour compatibilité avec Ocaml *)
function nbits   (a : xint)             : longint;            (* taille de a     *)
function lowbits (a : xint)             : longint;            (* voir Ocaml      *)
function highbits(a : xint)             : longint;            (* voir Ocaml      *)
function nth_word(a : xint; n : longint) : longint;           (* bits 16b..16b+15*)
function int_of  (a : xint)             : longint;            (* bits 0..30      *)

(* conversion en chaine *)
function string_of(x : xint) : ansistring;

(* chronométrage *)
procedure chrono(msg : pchar);

(* erreurs *)
procedure failwith(msg : pchar);
```

5.2 Précisions

Une variable de type `xint` est un pointeur sur une structure de données gérée par le gestionnaire mémoire intégré à la version Pascal de `Numerix`. L'initialisation d'une variable `x` de type `xint` s'effectue normalement en deux étapes :

1. initialisation du pointeur `x` par appel à `xnew` ;
2. affectation d'une valeur en résultat d'un calcul, par exemple `add(x,a,b)`.

Les fonctions `of_int`, `of_xint` et `of_string` réalisent à la fois l'initialisation du pointeur et l'affectation d'une valeur à partir d'un entier ordinaire, d'un autre grand entier ou d'une chaîne de caractères. L'instruction :

`x := of_int(`*i*`) ;`

est formellement équivalente aux instructions :

`x := xnew ; copy_int(x,`*i*`) ;`

et de même pour `of_xint` et `of_string`. L'exécution d'une opération par un appel tel que :

`quo(x,a,b) ;`

est valide si et seulement si les conditions suivantes sont remplies :

1. les variables `a` et `b` sont initialisées et désignent des valeurs valides pour l'opération en question (`a` désigne une valeur quelconque et `b` une valeur quelconque non nulle) ;
2. la variable `x` contient un pointeur initialisé.

Il n'est pas nécessaire que `x` ait reçu une valeur préalablement à cette opération. Si c'est le cas alors le bloc mémoire contenant cette valeur est surchargé avec le résultat de l'opération s'il est suffisamment grand, sinon un nouveau bloc mémoire est alloué pour recevoir le résultat, le pointeur `x` est modifié pour désigner ce nouveau bloc et l'ancien bloc est désalloué.

Les opérations de type lecture-modification-écriture où une même variable figure à la fois en positions opérande et résultat sont correctement traitées. Par contre pour les opérations produisant plusieurs résultats (`quomod`, `gcd_ex`, `cfrac` et `split`) une même variable ne peut figurer plusieurs fois en position résultat. Par exemple l'instruction suivante est illégale :

```
quomod(x,x,a,b); (* illégal *)
```

La procédure `xfree` permet de forcer la libération du bloc mémoire occupé par une valeur qui n'est plus utile. L'instruction :

```
xfree(x);
```

a pour effet de désallouer la valeur désignée par `x` s'il y en a une et de réinitialiser le pointeur `x`. La variable `x` reste utilisable après cette instruction pour recevoir une nouvelle valeur.

La conversion d'un grand entier en chaîne de caractères est réalisée par la fonction `string_of` qui alloue une chaîne de caractères de taille convenable et y copie la représentation décimale de son argument. Cette chaîne sera automatiquement libérée par le gestionnaire mémoire de Free Pascal lorsqu'elle ne sera plus référencée.

La procédure `chrono` affiche sur le canal d'erreur standard le temps CPU en secondes écoulé depuis le début du processus, la différence avec le temps précédent et le message passé en argument.

Enfin, la procédure `failwith` affiche le message passé en argument sur le canal d'erreur et arrête le programme.

5.3 Compilation

Les programmes Pascal utilisant `Numerix` doivent être compilés avec la commande suivante :

`ppc386` *options* `-Fu`*ppu_path* `-Fl`*lib_path* *fichiers source*

`-Fu`*ppu_path* désigne le répertoire contenant les fichiers compilés `clong.ppu`, `clong.o`, `dlong.ppu`, `dlong.o`, `slong.ppu` et `slong.o`. `-Fl`*lib_path* désigne le répertoire contenant la bibliothèque `libcnumx.a`. Ces directives peuvent être omises si ces fichiers sont stockés dans les répertoires normalement examinés par le compilateur Pascal.

5.4 Exemple

```
program simple;

 (* +-------------------------------------------------------------------------+
    |  Fichier simple.p: Démonstration de la bibliothèque Numerix avec Pascal  |
    +-------------------------------------------------------------------------+ *)

(* calcul de (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales *)

uses clong; (* ou dlong ou slong *)

procedure calcul(var res : xint; n : longint);
var dix_n, dix_2n, sqrt_2, sqrt_3, x, y : xint;
begin
   dix_n  := xnew;
   dix_2n := xnew;
   sqrt_2 := xnew;
   sqrt_3 := xnew;
   x      := xnew;
   y      := xnew;

   copy_int(dix_n,5);
   pow     (dix_n,  dix_n,  n);       (* dix_n <- 5^n ...       *)
   shiftl  (dix_n,  dix_n,  n);       (*        ... * 2^n       *)
   sqr     (dix_2n, dix_n);           (* dix_2n <- 10^(2n)      *)
   mul_1   (sqrt_2, dix_2n, 200);     (* sqrt_2 <- ...          *)
   sqrt    (sqrt_2, sqrt_2);          (* .. sqrt(2*10^(2n+2))  *)
   mul_1   (sqrt_3, dix_2n, 300);     (* sqrt_3 <- ...          *)
   sqrt    (sqrt_3, sqrt_3);          (* .. sqrt(3*10^(2n+2))  *)
   add     (x,      sqrt_3, sqrt_2);  (* x <- sqrt_3 + sqrt_2  *)
   sub     (y,      sqrt_3, sqrt_2);  (* y <- sqrt_3 - sqrt_2  *)
   mul     (x,      dix_n,  x);       (* res <- trunc(10^n*x...*)
   quo     (res,    x,      y);       (*              ... /y) *)

   xfree(dix_n);
   xfree(dix_2n);
   xfree(sqrt_2);
   xfree(sqrt_3);
   xfree(x);
   xfree(y);

end;


(* programme principal *)
var n,i : longint;
    c   : word;
    x   : xint;
begin
```

```
   n := 30;
   x := xnew;
   i := 1;
   while i <= paramcount do begin
      if  paramstr(i) = '-test' then n := 30
      else if (paramstr(i) = '-n') and (i < paramcount) then begin
         i := i+1;
         val(paramstr(i),n,c);
      end
      else failwith('syntaxe : simple [-test] [-n <n>]');
      i := i+1;
   end;

   calcul(x,n);
   writeln(string_of(x));

   xfree(x);

end.
```

Compilation et exécution :

```
> ppc386 -Fu/home/quercia/lib -Fl/home/quercia/lib simple.p
Free Pascal Compiler version 1.0.2 [2001/03/12] for i386
Copyright (c) 1993-2000 by Florian Klaempfl
Target OS: Linux for i386
Compiling simple.p
Assembling simple
Linking simple
69 Lines compiled, 0.1 sec
> ./simple
98989794855663561963945681494 10
```

Chapitre 6

Installation

6.1 Téléchargement

`Numerix` est disponible à l'URL suivant :

`http://pauillac.inria.fr/~quercia/cdrom/bibs/numerix.tar.gz`

Vous devez disposer du compilateur C `gcc` pour compiler les parties C et assembleur de la bibliothèque. A priori toute version récente de `gcc` devrait convenir, la bibliothèque a été compilée avec succès sur un PC-Linux avec `egcs-2.91.66` ainsi que sur une station Dec avec `gcc 2.7.2.3` pour le module `Clong` uniquement.

Pour Ocaml vous devez disposer d'une version supérieure ou égale à `1.05` et pour Camllight d'une version supérieure ou égale à `0.73`. Ocaml et Camllight sont disponibles à l'URL :

`http://pauillac.inria.fr/caml/index-fra.html`

Si vous voulez inclure dans les interfaces Ocaml et Camllight le module `Gmp` vous devez disposer de `GMP` qui est téléchargeable à l'URL :

`http://www.swox.com/gmp/`

L'interface Pascal ne peut être compilée que sur un PC-Linux avec le compilateur Free Pascal téléchargeable à l'URL :

`http://www.freepascal.org/`

Notez que l'interface Pascal utilise la version compilée de l'interface C : `libcnumx.a`. Par conséquent l'interface Pascal ne peut être compilée que si l'interface C l'est aussi.

6.2 Configuration automatique

Extraire l'archive `numerix.tar.gz` dans un répertoire temporaire et lancer le script de configuration qui se trouve à la racine.

Sur un PC-Linux avec le shell `bash` lancez :

```
./configure   2>&1 | tee conflog
```

Sur une station Dec avec le shell `csh` lancez :

```
./configure       |& tee conflog
```

Ce script détermine quelles parties de `Numerix` peuvent être compilées sur votre machine et crée un fichier `Makefile` et un fichier `numconf` correspondant à votre configuration. Le script `configure` reconnaît les options suivantes :

`- -prefix=`*dir*

Fixe la racine commune aux répertoires d'installation :
`INSTALL_LIB     =` *dir*`/lib`,
`INSTALL_BIN     =` *dir*`/bin`,
`INSTALL_INCLUDE =` *dir*`/include`.
Le préfixe par défaut est `$HOME`.

`--libdir=`*dir,* `--bindir=`*dir,* `--includedir=`*dir*

Fixe l'un des répertoire `INSTALL_LIB`, `INSTALL_BIN` et `INSTALL_INCLUDE` indépendamment des autres.

`--enable-c`, `--disable-c`

Sélectionne ou désélectionne l'interface C. Par défaut elle est sélectionnée.

`--enable-caml`, `--disable-caml`

Sélectionne ou désélectionne l'interface Camllight. Par défaut elle est sélectionnée si Camllight a été trouvé sur la machine.

`--enable-ocaml`, `--disable-ocaml`

Sélectionne ou désélectionne l'interface Ocaml. Par défaut elle est sélectionnée si Ocaml a été trouvé sur la machine.

`--enable-pascal`, `--disable-pascal`

Sélectionne ou désélectionne l'interface Pascal. Par défaut elle est sélectionnée si `ppc386` a été trouvé sur la machine.

`--enable-clong`, `--disable-clong`

Sélectionne ou désélectionne le module `Clong`. Par défaut il est sélectionné.

`--enable-dlong`, `--disable-dlong`

Sélectionne ou désélectionne le module `Dlong`. Par défaut il est sélectionné si le compilateur C reconnaît le type entier double précision (`long long`) et si ce type est effectivement de longueur double de celle d'un `long`.

`--enable-slong`, `--disable-slong`

Sélectionne ou désélectionne le module `Slong`. Par défaut il est sélectionné si le processeur est de type Intel x86 et le type `long long` est correctement implémenté par le compilateur C.

--enable-gmp, --disable-gmp

Sélectionne ou désélectionne le module Gmp. Par défaut il est sélectionné si la bibliothèque GMP a été trouvée sur la machine.

--enable-caml_bignum, --disable-caml_bignum
--enable-ocaml_bignum, --disable-ocaml_bignum

Sélectionne ou désélectionne le module Big séparément pour Camllight et pour Ocaml. Par défaut ce module est sélectionné si la bibliothèque libnums associée a été trouvée.

--disable-lang

Désélectionne tous les langages non explicitement sélectionnés par une option --enable-xxx.

--disable-modules

Désélectionne tous les modules non explicitement sélectionnés par une option --enable-xxx.

--disable-all

Désélectionne toutes les combinaisons de langage et modules non explicitement sélectionnées par une option --enable-xxx.

6.3 Édition du fichier numconf

Normalement le script configure décrit à la section précédente devrait créer un fichier numconf convenable. En cas de problème vous pouvez éditer ce fichier et corriger les valeurs indiquées par configure si elles sont incorrectes. Si vous modifiez numconf, il faut recréer un fichier Makefile pour prendre en compte les modifications effectuées. Pour ce faire lancez la commande :

```
make -f numconf makerules
```

La plupart de ces paramètres renseignés dans le fichier numconf doivent recevoir une valeur booléenne avec la convention 0 = faux et 1 = vrai. A titre d'exemple, les fichiers logs/pentium/numconf et logs/alpha/numconf contiennent les réglages ayant servi pour les tests sur processeurs Pentium et Digital alpha.

```
BITS = 32
```

Indiquez la taille d'un mot machine, les seules valeurs reconnues sont 32 et 64.

```
USE_CLONG        = 1
USE_DLONG        = 1
USE_SLONG        = 1
USE_GMP          = 1
USE_CAML_BIGNUM  = 1
USE_OCAML_BIGNUM = 1
```

Indiquez les modules que vous voulez compiler, plusieurs modules peuvent être sélectionnés. Les modules Dlong et Slong ne peuvent être compilés que pour les machines équipées d'un processeur Intel x86. Les modules Gmp et Big ne peuvent être compilés que si vous disposez de GMP et BigNum.

```
MAKE_C_LIB      = 1
MAKE_OCAML_LIB  = 1
MAKE_CAML_LIB   = 1
MAKE_PASCAL_LIB = 1
```

Indiquez quelles interfaces vous voulez compiler.

```
GCC = gcc -O2 -Wall
AR  = ar -rc
RANLIB = ranlib
```

Indiquez les commandes de compilation C et de création d'archives. Vous pouvez au besoin ajouter des directives `-Ixxx` et `-Lxxx` si le compilateur ou l'éditeur de liens ne trouvent pas par eux-mêmes certains fichiers d'en-tête ou certaines bibliothèques

```
OCAML_VERSION = 204
OCAML_LIBDIR  = /usr/local/lib/ocaml
CAML_VERSION  = 074
CAML_LIBDIR   = /usr/local/lib/caml-light
```

Indiquez sans point décimal les versions de Ocaml et Camllight dont vous disposez et les répertoires où sont installées ces versions. Ces variables ne sont consultées que si les interfaces Ocaml et Camllight doivent être compilées.

```
CAML       = camlc
CAMLLIBR   = camllibr
CAMLMKTOP  = camlmktop

OCAML      = ocamlc
OCAMLOPT   = ocamlopt
OCAMLMKTOP = ocamlmktop
```

Indiquez les commandes que `make` doit lancer pour invoquer respectivement le compilateur Camllight, l'archiveur Camllight, le compilateur de toplevel Camllight, le compilateur Ocaml, le compilateur optimiseur Ocaml et le compilateur de toplevel Ocaml. Normalement les chaînes ci-dessus doivent convenir, éventuellement avec une indication du chemin d'accès.

```
PPC = ppc386
```

Indiquez la commande que `make` doit lancer pour invoquer le compilateur Pascal. Normalement la chaîne ci-dessus doit convenir, éventuellement avec des indications de chemins d'accès.

```
INSTALL_LIB     = $(HOME)/lib
INSTALL_INCLUDE = $(HOME)/include
INSTALL_BIN     = $(HOME)/bin
```

Indiquez dans quels répertoires doivent être installés les bibliothèques compilées, les fichiers d'en-tête (pour l'interface C uniquement) et les programmes exécutables. La variable `INSTALL_LIB` est recopiée en dur dans les systèmes interactifs `ocamlnumx` pour Ocaml $\geqslant$ 1.07 et `camlnumx` pour Camllight $\geqslant$ 0.74

de façon que ces systèmes puissent trouver par eux-mêmes les interfaces compilées `numerix.cmi` et `numerix.zi`. Les versions antérieures de Ocaml et Camllight ne disposent pas de cette facilité et vous devrez indiquer par une directive `directory` l'emplacement de ces interfaces compilées à chaque lancement du système interactif. De même l'activation des pretty-printers n'est faite automatiquement qu'à partir de Ocaml-1.07 et Camllight-0.74 ; pour les versions antérieures vous devrez l'effectuer manuellement avec la directive `install_printer`.

6.4 Modification de la bibliothèque Bignum

A partir de la version Ocaml-3.00 les données externes à Ocaml peuvent être comparées, hachées, exportées et importées via les fonctions génériques de Ocaml, sous réserve de fournir des interfaces avec ces fonctions génériques. Les modules `Clong`, `Dlong`, `Slong` et `Gmp` comportent de telles interfaces, mais le module `Big_int`, livré en standard avec Ocaml, ne comporte que les interfaces avec les fonctions d'exportation et d'importation, il manque les interfaces avec les fonctions de comparaison et de hachage.

Le répertoire `lib/ocaml` de `Numerix` contient un fichier correctif fournissant ces interfaces manquantes, `bignum.patch`, il peut être utilisé pour modifier la bibliothèque `Bignum` afin de disposer des comparaisons et du hachage générique avec le module `Big`.

Note : les modifications décrites ci-dessous sont facultatives. Si vous ne comptez pas utiliser le module `Big`*, ou si vous pouvez l'utiliser sans les comparaisons et le hachage génériques, il est déconseillé de procéder à ces modifications ; vous pouvez passer dans ce cas à la section suivante :* **Compilation**.

Vous devez disposer des sources Ocaml. Décompactez les dans un répertoire temporaire, placez vous à la racine de l'arborescence créée (`ocaml-`*x.yy*) et lancez la commande :

```
patch -b -p0 <.../numerix-0.19/lib/ocaml/bignum.patch
```

où `...` désigne le répertoire contenant les sources de `Numerix`. Le programme `patch` doit rendre compte de la correction avec un message tel que :

Sur un PC Linux

```
patching file otherlibs/num/nat_stubs.c
```

Sur une Station Dec

```
Hmm...  Looks like a new-style context diff to me...
The text leading up to this was:
--------------------------
|*** otherlibs/num/nat_stubs.c.orig     Fri Aug 24 15:08:51 2001
|--- otherlibs/num/nat_stubs.c  Fri Aug 24 15:46:58 2001
--------------------------
```

```
Patching file otherlibs/num/nat_stubs.c using Plan A...
Hunk #1 succeeded at 11.
Hunk #2 succeeded at 30.
Hunk #3 succeeded at 335.
done
```

S'il n'y a pas d'erreur, vous pouvez recompiler et installer le système Ocaml modifié.

6.5 Compilation

Après configuration et adaptation éventuelle du fichier `numconf` vous pouvez lancer la compilation. Les cibles sont :

`lib` :
: compile les bibliothèques et les fichiers d'interface ;

`exemples` :
: compile les exemples ;

`test` :
: exécute chaque programme exemple avec l'option `-test` ;

`install` :
: copie les bibliothèques, les fichiers d'en-tête et les exécutables dans les répertoires désignés par les variables `INSTALL_xxx` ;

`clean` :
: supprime tous les fichiers compilés.

Sur un PC-Linux avec le shell `bash` lancez dans cet ordre :

```
make lib      2>&1 | tee liblog
make exemples 2>&1 | tee exlog
make test     2>&1 | tee testlog
```

Sur une station Dec avec le shell `csh` lancez dans cet ordre :

```
make lib      |& tee liblog
make exemples |& tee exlog
make test     |& tee testlog
```

Il ne doit y avoir aucune erreur de compilation ni aucun message d'avertissement. S'il y en a et si vous n'arrivez pas à débloquer la situation envoyez par courrier électronique les fichiers de trace `liblog`, `exlog` et `testlog` à `michel.quercia@prepas.org` pour analyse. Si vous avez rencontré des problèmes que vous avez pu résoudre par vous même, merci de me le faire savoir afin que je puisse corriger les fichiers fautifs. Les fichiers `logs/pentium/xxxlog` et `logs/alpha/xxxlog` inclus dans la distribution contiennent les traces de compilation sur un PC-Linux et sur une station Dec, vous pouvez vous y référer en cas de problème.

Si la compilation et les tests se sont déroulés correctement vous pouvez installer la bibliothèque `Numerix` avec l'une des commandes :

```
make install  2>&1 | tee inslog
make install  |& tee inslog
```

Ceci a pour effet de copier les fichiers :

```
big.zi       dlong.o      infgmp.zi    numcount.cmi slong.ppu
camlnumx     dlong.zi     infslong.zi  numerix.a    slong.o
clong.ppu    gmp.zi       libclnumx.a  numerix.cma  slong.zi
clong.o      infbig.zi    libcnumx.a   numerix.cmi
clong.zi     infclong.zi  libmlnumx.a  numerix.cmxa
dlong.ppu    infdlong.zi  numcmp.cmi   numerix.zo
```

dans le répertoire `$(INSTALL_LIB)`,

```
c-long_int.h  long_int.h
```

dans le répertoire `$(INSTALL_INCLUDE)` et

```
ocamlnumx
```

dans le répertoire `$(INSTALL_BIN)` ou seulement certains de ces fichiers selon les modules et interfaces que vous avez sélectionné.

L'installation est terminée, vous pouvez à présent vous adonner aux joies du calcul multiprécision. Le guide d'utilisation que vous lisez présentement est disponible dans le sous-répertoire `doc` au format Postscript (`numerix.ps`) et au format LaTeX (`numerix.tex`) si vous désirez l'imprimer.

6.6 Description des exemples

Le sous-répertoire `exemples` contient divers programmes utilisant `Numerix` en C, Camllight, Ocaml ou Pascal. Chaque exemple est placé dans un répertoire distinct avec des sous-répertoires `c`, `caml`, `ocaml` et `pascal` contenant les fichiers source correspondant.

Les programmes écrits en C sont fournis en deux versions : une utilisant la bibliothèque `Numerix` (fichier `xxx-long.c`) et une utilisant la bibliothèque `GMP` (fichier `xxx-gmp.c`). Les exécutables produits ont un nom de la forme : `xxx-c`, `xxx-d`, `xxx-s` et `xxx-g`, la dernière lettre indiquant quel module est utilisé.

Les programmes écrits en Camllight comportent un fichier source `xxx.mlp` qui après compilation donne les exécutables `xxx-b`, `xxx-c`, `xxx-d`, `xxx-s` et `xxx-g`, la dernière lettre indiquant quel module est utilisé.

Les programmes écrits en Ocaml comportent un seul fichier source `xxx.ml` qui après compilation donne les exécutables `xxx-byte` (produit par le compilateur `ocamlc`) et `xxx-opt` (produit par le compilateur `ocamlopt`). Chacun de ces exécutables contient tous les modules grands entiers compilés et un module particulier est sélectionné à l'exécution par une option `-e` *module* comme décrit à la section 2.2.

Les programmes écrits en Pascal comportent un fichier source `xxx.p` qui après compilation donne les exécutables `xxx-c`, `xxx-d` et `xxx-s`, la dernière lettre indiquant quelle unité est utilisée.

6.6.1 chrono

Mesure de vitesse des différentes bibliothèques (source en C uniquement). Ce programme tire au hasard des grands entiers de n et $2n$ bits et chronomètre le temps de diverses opérations entre ces entiers :

`mul`	: multiplication n bits par n bits ;
`sqr`	: carré d'un nombre de n bits ;
`quomod`	: division avec reste $2n$ bits par n bits ;
`quo`	: division sans reste $2n$ bits par n bits ;
`sqrt`	: racine carrée d'un entier de $2n$ bits ;
`gcd`	: pgcd de deux entiers de n bits ;
`gcd_ex`	: pgcd et coefficients de Bézout de deux entiers de n bits ;
`all`	: toutes les opérations ci-dessus.

Indiquez sur la ligne de commande la valeur de n et quelles opérations effectuer parmi `mul`, `sqr`, `quomod`, `quo`, `sqrt`, `gcd` et `gcd_ex`. Vous pouvez indiquer un facteur de répétition par l'option `-r` r, dans ce cas chaque opération est répétée r fois. Par exemple :

```
> exemples/chrono/c/chrono-g -all 1000000
    0.11     0.11 début
    1.15     1.04 mul
    1.84     0.69 sqr
    6.02     4.18 quomod
   10.23     4.21 quo
   40.09    29.86 sqrt
  242.48   202.39 gcd
  698.63   456.15 gcd_ex
> exemples/chrono/c/chrono-s -all 1000000
    0.23     0.23 début
    1.11     0.88 mul
    1.65     0.54 sqr
    6.50     4.85 quomod
   10.62     4.12 quo
   13.93     3.31 sqrt
   45.59    31.66 gcd
   86.28    40.69 gcd_ex
```

Donc sur la machine de test (Pentium-233Mhz) avec le module `Gmp-3` une multiplication entre deux nombres d'un million de bits prend 1.04 secondes, le carré d'un nombre d'un million de bits prend 0.69 secondes, etc. Le second test présente les temps correspondants pour le module `Slong`.

6.6.2 hamming

Calcul du k-ème entier de Hamming ou du rang d'un entier de Hamming (source en Ocaml uniquement). Le problème de Hamming et divers algorithmes le résolvant sont présentés dans les documents :

```
http://pauillac.inria.fr/~quercia/papers/hamming.ps.gz
http://pauillac.inria.fr/~quercia/papers/hamming-new.ps.gz
```

Le programme donné en exemple avec `Numerix` implémente un algorithme non publié de complexité $O(k^{1/3}\ln(k))$ (`hamming-new` implémente un algorithme plus rapide en $O(\ln(k)^3)$). La syntaxe de la ligne de commande est :

`hamming-xxx (`*k* `|` *x y z*`) [-d] [-v` *volume*`] [-e` *entiers*`]`

où k désigne le rang de l'entier cherché et x, y, z les exposants de $2, 3, 5$ d'un entier de Hamming dont on cherche le rang. `-d` demande d'afficher les différentes phases du programme avec chronométrage, `-v` *volume* désigne la méthode de calcul des volumes de tétraèdres (`surplace` ou `func`) et `-e` *entiers* le module des grands entiers à utiliser. Par exemple :

```
> exemples/hamming/ocaml/hamming-opt -e slong -d -v func 1T
    0.03     0.03 entiers    Slong
    0.03     0.00 volume     func
    0.03     0.00 Paramètres 12081378335  14699735488
    0.03     0.00            19148531618  23298529519
    0.03     0.00            28052091781  34131728817
    0.03     0.00 Approx     338940570505262 -> 1000267456452
    0.03     0.00 Approx     338910358662417 -> 1000000023838
    0.03     0.00 Approx     338910355969313 -> 1000000000001
    0.37     0.34 Exact      338910355969313 -> 1000000000001
    0.40     0.03 n2         338910355969346
    0.42     0.02 n0         338910355969286
    0.50     0.08 n1         338910355969190
    0.51     0.01 Dichotomie 8295 2426 6853
x = 8295  y = 2426  z = 6853
```

6.6.3 nextpp

Recherche d'entiers pseudo-premiers avec le test de Rabin-Miller (source en Caml, Ocaml, C et Pascal). Ce programme prend deux entiers n et a dans cet ordre sur la ligne de commande et recherche le plus petit entier à $n + 1$ chiffres pseudo-premier en base a. Par exemple :

```
> time exemples/nextpp/c/nextpp-s 1000 2
453

real    0m32.111s
user    0m31.980s
sys     0m0.000s
```

```
> time exemples/nextpp/c/nextpp-g 1000 2
453

real    0m30.184s
user    0m30.050s
sys     0m0.010s
```

ce qui signifie que $10^{1000} + 453$ est le plus petit entier à 1001 chiffres pseudo-premier en base 2. Ce programme est le seul parmi tous les exemples où `Gmp-3` se révèle plus rapide que `Slong`, du fait d'une plus grande vitesse de calcul pour l'exponentiation modulaire.

6.6.4 pi

Calcul des n premières décimales de π (sources en C, Camllight, Ocaml et Pascal). Ce programme implémente le calcul approché de π décrit dans le manuel de référence de la bibliothèque `BigNum` (*The Caml Numbers Reference Manual*, Inria, RT-0141) avec une technique de sommation dichotomique pour accélérer le calcul de la série (les termes sont additionnés deux par deux, puis 4 par 4, puis 8 par 8 etc). Par exemple :

```
> exemples/pi/ocaml/pi-opt -e slong -d 1000000 -skip
    0.03     0.03 Slong, surplace
    1.67     1.64 puiss 5
   19.75    18.08 sqrt
  122.44   102.69 série lb=6875847 ls=6875855
  150.43    27.99 quotient
  186.94    36.51 conversion
3.
14159 26535  89793 23846  26433 83279  50288 41971  69399 37510
... (19998 lignes omises)
56787 96130  33116 46283  99634 64604  22090 10610  57794 58151
```

6.6.5 simple

Programme de démonstration élémentaire (sources en C, Camllight, Ocaml et Pascal). Ce programme sert de support à la présentation des interfaces de `Numerix`. Il calcule les n premières décimales du nombre $(\sqrt{3}+\sqrt{2})/(\sqrt{3}-\sqrt{2})$. Par exemple :

```
> exemples/simple/ocaml/simple-opt -e slong -n 50
9898979485566356196394568149411782783931894961313 35
```

Chapitre 7

Informations techniques

7.1 Algorithmes

7.1.1 Multiplication

`Numerix` implémente quatre algorithmes de multiplication chacun adapté à une certaine plage d'opérandes.

`mul_2` : multiplication d'un entier long par un entier simple ou double précision. Cette multiplication est effectuée par une simple boucle avec une complexité $O(n)$, n étant la taille de l'entier long.

`mul_n2` : multiplication de deux entiers longs, le plus court ayant au plus `klim` chiffres. Cette multiplication est effectuée par une double boucle de complexité $O(np)$ où n et p sont les tailles des opérandes.

`karamul` : multiplication de deux entiers longs ayant entre `klim` et `flim` chiffres par l'algorithme « diviser pour régner » de Karatsuba. Soient $n \leqslant p$ les tailles des opérandes : si $n \leqslant p/2$ on divise l'opérande de taille p en deux blocs de tailles n et $p - n$ et on calcule récursivement les deux produits correspondants ($n \times n$ et $n \times (p - n)$). Lorsque $n = p$ la complexité de cette multiplication est $O(n^{\ln(3)/\ln(2)})$.

`fftmul` : multiplication de deux entiers longs de tailles n, p avec `klim` $< n \leqslant p$ et $p \geqslant$ `flim` par l'algorithme de transformée de Fourier rapide modulaire de Schönage. Les opérandes et le produit sont considérés comme des polynômes à coefficients dans $\mathbb{Z}/(2^{2^\ell} + 1)\mathbb{Z}$, le polynôme produit étant calculé à partir de ses valeurs aux points 2^k, $0 \leqslant k < 2^{\ell+1}$ qui forment un sous-groupe multiplicatif cyclique de $(\mathbb{Z}/(2^{2^\ell} + 1)\mathbb{Z})^*$. Le paramètre ℓ est déterminé en fonction de la taille du *nombre* produit de façon à minimiser une estimation approximative du temps de calcul. Typiquement $5 \leqslant \ell \leqslant 15$. La complexité théorique de l'algorithme de Schönage est $O(n\ln(n))$ pour un produit de taille n si ℓ peut être stocké sur un mot machine, $O(n\ln(n)\ln(\ln n))$ sinon (cas inenvisageable avec les machines actuelles). Cependant la méthode de choix de ℓ ne permet pas d'estimer facilement la complexité de l'implémentation, expérimentalement le temps de calcul est presque linéaire jusqu'à une dizaine de millions de bits, au delà apparaissent des problèmes dûs à la quantité importante de mémoire auxiliaire consommée.

Les valeurs de `klim` et `flim` ont été déterminées par tâtonnements. Pour une architecture 32 bits avec des chiffres de 32 bits, on a `klim` = 32 chiffres, `flim` = 937 chiffres c'est-à-dire que l'on bascule de la multiplication naïve en $O(n^2)$ à celle de Karatsuba à partir de 1024 bits et de la multiplication de Karatsuba à celle de Schönage à partir de 30000 bits.

Les algorithmes de multiplication naïve et de Karatsuba sont décrits dans de de nombreux ouvrages, notamment *Knuth, The art of computer programming, vol. 2*. La multiplication par FFT modulaire de Schönage est décrite dans un article de Paul Zimmermann :

`http://www.loria.fr/~zimmerma/bignum/schon.ps.gz`

Ces algorithmes sont également utilisés pour l'élévation au carré, en tenant compte des simplifications dues au fait que les deux opérandes à multiplier sont égaux.

7.1.2 Division

`Numerix` implémente trois algorithmes de division chacun adapté à une certaine plage d'opérandes.

- `quo_2` : division d'un entier long par un entier simple ou double précision, par une simple boucle de complexité $O(n)$ où n est la taille du dividende.
- `quo_n2` : division de deux entiers longs par l'algorithme « scolaire » lorsque la taille du diviseur ou celle du quotient est inférieure à `klim`. Cette division a pour complexité $O(np)$ où n est la taille du diviseur et p celle du quotient.
- `bzdiv` : division de deux entiers longs par l'algorithme de Burnikel et Ziegler, qui est une version « diviser pour régner » de l'algorithme scolaire. Avec cet algorithme le temps nécessaire pour diviser un nombre de $2n$ bits par un nombre de n bits est de l'ordre de deux multiplications n bits par n bits dans la plage de la multiplication de Karatsuba, et de l'ordre de $C\ln(n)$ multiplications n bits par n bits dans la plage de la multiplication par FFT. En pratique, jusqu'à une dizaine de millions de bits, le temps d'une division $2n$ par n bits est au plus égal à celui de cinq multiplications n bits par n bits.

Le calcul de la racine carrée entière d'un entier long est réalisé par des algorithmes similaires : pour calculer $b = \lfloor\sqrt{a}\rfloor$ on divise a par b, le quotient est à deux unités près le nombre b cherché. Pour effectuer cette division, il suffit de connaître le chiffre de tête de b, les chiffres suivants sont déterminés en cours de calcul, c'est-à-dire que l'on découvre le diviseur au fur et à mesure que la division se déroule. Le chiffre de tête de b est déterminé indépendamment par itération de Newton. Le temps nécessaire au calcul de la racine carrée d'un nombre de $2n$ bits selon cette méthode est théoriquement de l'ordre de la moitié du temps d'une division $2n$ bits par n bits, soit équivalent à une multiplication n bits par n bits dans la plage de la multiplication de Karatsuba (rapport effectivement observé), et $C\ln(n)/2$ dans la plage de la multiplication par FFT (temps constaté pour 10^7 bits : 3 multiplications).

La conversion d'un entier long en décimal peut être effectuée par divisions successives par 10, mais cela conduit à un algorithme de complexité $O(n^2)$ pour convertir un nombre de n bits. La méthode utilisée par `Numerix` consiste à calculer, par élévations au carré successives, une puissance de 10 : 10^{2^k} dont la longueur est comprise entre $n/2$ et n bits puis à diviser le nombre à convertir par 10^{2^k} et à convertir récursivement le quotient et le reste à l'aide des puissances de 10 déjà calculées. De la sorte le temps de conversion est de l'ordre de deux multiplications n bits par n bits dans la plage de la multiplication de Karatsuba et $O(\ln(n))$ multiplications n bits par n bits dans la plage de la multiplication par FFT.

Références

Division scolaire : *Knuth, The art of computer programming, vol. 2*

Division de Burnikel et Ziegler : *Fast recursive division, Rapport de recherche MPI-I-98-1-022, Max Planck Institut Für Informatik, octobre 1998*

Racine carrée récursive (Zimmermann) : *Karatsuba square root, Rapport de recherche INRIA n^o^ 3805,* `http ://www.inria.fr/RRRT/RR3805.html`

Versions itératives des algorithmes de Burnikel-Ziegler et Zimmermann :
Division et racine carrée rapides,
`http://pauillac.inria.fr/~quercia/papers/divrap.ps.gz`

7.1.3 Pgcd

Le calcul du pgcd de deux entiers a fait l'objet de moins d'efforts d'optimisation que les algorithmes précédents, un seul algorithme est implémenté : une version « diviser pour régner » de l'algorithme de Lehmer.

L'algorithme de Lehmer, décrit dans le livre de Knuth, pour calculer le pgcd et les coefficients de Bézout de deux entiers longs a et b consiste à développer en fraction continue le nombre a/b, les quotients obtenus étant ceux de l'algorithme d'Euclide appliqué à a et b. Les quotients sont déterminés à partir des chiffres de tête α et β de a et b en développant en parallèle en fraction continue $\alpha/(\beta+1)$ et $(\alpha+1)/\beta$ qui encadrent a/b tant que les quotients sont identiques pour les deux fractions encadrantes. Lorsque l'on constate une différence, on applique la transformation homographique obtenue à a et b puis on itère la méthode. Cet algorithme a la même complexité asymptotique que l'algorithme d'Euclide, soit $O(n^2)$ pour des opérandes de n bits, mais la plupart des opérations sont effectuées en simple précision et il se révèle en pratique environ 20 fois plus rapide que l'algorithme d'Euclide sur des grands nombres.

`Numerix` implémente une version « diviser pour régner » de la méthode de Lehmer en prenant pour α et β les moitiés hautes de a et b ce qui donne une complexité théorique de 18 multiplications n bits par n bits dans la plage de la multiplication de Karatsuba (confirmée expérimentalement) et $O(\ln(n))$ multiplications n bits par n bits dans la plage de la multiplication par FFT (pour 10^6 bits, le rapport est de l'ordre de 45). Cette version récursive de l'algorithme de Lehmer n'est décrite dans aucun des documents que j'ai consultés, par contre Schönage a donné un algorithme apparenté, appelé *demi-pgcd* ou *descente controlée*, décrit dans :

> *Equation Solving in Terms of Computational Complexity,*
> *Proceedings of the International Congress of Mathematicians, 1987*

7.2 Différences entre `Numerix`-0.13 et `Numerix`-0.17

`Numerix`-0.13 était la première version publique, diffusée en février 2000. Les versions suivantes 0.14, 0.15 et 0.16 ont réglé au coup par coup quelques problèmes de compilation signalés par certains utilisateurs, et cette version, `Numerix`-0.17, est la deuxième véritable version publique.

Noyau :

Réécriture complète du noyau C et assembleur de façon à permettre une évolution ultérieure plus facile.

Correction d'un bogue dans le code de la division rapide (dépassement de capacité dans une soustraction, signalé par YAMAGATA yoriyuki).

Implémentation de la multiplication de Schönage.

Interface Caml/Ocaml :

Mise à jour pour Ocaml-3.00.

Ajout de la racine p-ème entière (algorithme de Newton récursif).

Ajout de l'exponentiation modulaire.

Le module `Cmp` a été séparé de `Numerix` pour devenir un module indépendant : `Numcmp`.

Ajout du module `Numcount`.

Regroupement du code compilé de `Numerix`, `Numcmp` et `Numcount` dans les archives `numerix.cma`, `numerix.cmxa` au lieu de `numerix.cmo`, `numerix.cmx`.

7.3 Différences entre `Numerix`-0.17 et `Numerix`-0.18

Noyau :

Correction d'un bogue dans la division par un nombre de deux chiffres.

Codage de l'exponentiation modulaire en C, elle reste moins rapide que celle de `GMP` dans le cas où la base est un grand nombre.

Adaptation des règles de compilation pour OpenBSD-2.8.

Interface C :

Ajout de la fonction `root`.

Interface Pascal :

Première publication de l'interface Pascal.

7.4 Différences entre `Numerix`-0.18 et `Numerix`-0.19

Interface Ocaml :

Suppression des opérateurs de comparaison entre grands entiers dans le foncteur `Infixes`, les opérateurs de comparaison polymorphes de Ocaml deviennent utilisables avec les grands entiers.

Implémentation des fonctions de hachage, de sérialisation et de désérialisation.