Numerix

Bibliothèque de calcul en grands entiers

Michel Quercia
michel.quercia@prepas.org

version 0.21 du 18 mars 2005

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. Deux implémentations différentes du module `Slong` sont disponibles : une utilisant les instructions arithmétiques standard présentes sur tous les processeurs de la famille `x86`, et une utilisant les instructions `SSE2` disponibles à partir du Pentium-4.

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 Pentium-4-2.8Ghz avec 512Mo de mémoire vive :

	n	Slong SSE2	Slong x86	Dlong	Clong	GMP 4.1.4	Big_int
Numerix-0.21	10^4	0.01s	0.01s	0.04s	0.06s	0.01s	0.30s
	10^5	0.23s	0.46s	0.99s	1.25s	0.47s	29.75s
	10^6	4.24s	9.23s	18.91s	24.31s	11.33s	2830s
Numerix-0.19	10^4		0.03s	0.08s	0.08s		
	10^5		0.80s	1.92s	2.09s		
	10^6		16.24s	42.32s	47.91s		

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 `Clong`, `Dlong`, `Slong` 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.

Le tableau précédent donne aussi les temps de calcul pour le même programme et sur la même machine pour la version précédente de `Numerix`, publiée en 2002. Entre ces deux versions le code C du noyau a été entièrement réécrit avec des algorithmes plus sophistiqués, notamment pour la multiplication et la division des entiers de grande longueur, ce qui donne un facteur d'accélération de l'ordre de 2 pour les calculs sur des très grands nombres. Par ailleurs, l'utilisation du jeu d'instructions `SSE2` en mode `Slong` apporte un deuxième facteur d'accélération de l'ordre de 2 par rapport au jeu d'instructions arithmétiques standard des processeurs de type Pentium. Ainsi, la version la plus rapide de `Numerix`-0.21 est environ 4 fois plus rapide que celle de `Numerix`-0.19 pour ce test.

Chapitre 2

Utilisation avec Ocaml

Sommaire

2.1 Interface

`Numerix`-0.21 a été développé et testé avec Ocaml-3.08. Quelques essais avec Ocaml-3.06, Ocaml-3.07 et Ocaml-3.09-dev n'ont pas posé de problème particulier. Il est donc probable que `Numerix` fonctionne correctement pour toute version de Ocaml comprise au sens large entre 3.06 et 3.09. Le module `Numerix` contient :

- une description abstraite (signature `Int_type`) commune à toutes les implémentations des grands entiers disponibles ;
- les descriptions concrètes des sous-modules `Clong`, `Dlong`, `Slong`, `GMP` et `Big` conformes à la signature `Int_type` ;
- un foncteur `Infixes` permettant d'utiliser les opérations les plus courantes sur les grands entiers en notation infixe ;
- un foncteur `Cmp` produisant une nouvelle implémentation conforme à la signature `Int_type` à partir de deux telles implémentations `A` et `B`, et permettant de vérifier qu'un même calcul produit un résultat identique avec `A` et avec `B` ;
- un foncteur `Count` produisant une nouvelle implémentation conforme à la signature `Int_type` à partir d'une telle implémentation `A`, et permettant de tenir à jour des statistiques sur le nombre d'opérations entre grands entiers effectuées ainsi que sur les tailles moyenne et maximale des opérandes ;
- un foncteur `Rfuns` qui implémente des algorithmes d'approximations pour les fonctions mathématiques à valeurs réelles usuelles ;
- un foncteur `Start` permettant de sélectionner à l'exécution par une option en ligne de commande quelle implémentation des grands entiers utiliser ;
- une fonction de chronométrage.

Voici l'interface publique extraite du fichier `numerix.mli` :

```
                    (* +-------------------------+
                       |  Description abstraite  |
                       +-------------------------+ *)

(* mode d'arrondi *)
type round_mode = Floor | Nearest_up | Ceil | Nearest_down

module type Int_type = sig

  type t                     (* entier         *)
  type tref                  (* entier mutable *)
  val name : unit -> string  (* nom du module  *)
  val zero : t               (* le nombre 0    *)
  val one  : t               (* le nombre 1    *)

  (* référence      ------------------------------------------------- *)
  (*                mode          r       s       a    b      c      res  *)
  val make_ref    :                                t ->              tref
  val copy_in     :              tref ->           t ->              unit
  val copy_out    :              tref ->                             t
  val look        :              tref ->                             t
```

```
(* addition       ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val add         :                                t -> t   ->       t
val add_1       :                                t -> int ->       t
val add_in      :              tref ->           t -> t   ->       unit
val add_1_in    :              tref ->           t -> int ->       unit

(* soustraction   ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val sub         :                                t -> t   ->       t
val sub_1       :                                t -> int ->       t
val sub_in      :              tref ->           t -> t   ->       unit
val sub_1_in    :              tref ->           t -> int ->       unit

(* multiplication ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val mul         :                                t -> t   ->       t
val mul_1       :                                t -> int ->       t
val mul_in      :              tref ->           t -> t   ->       unit
val mul_1_in    :              tref ->           t -> int ->       unit

(* division       ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val quomod      :                                t -> t   ->       t*t
val quo         :                                t -> t   ->       t
val modulo      :                                t -> t   ->       t
val gquomod     : round_mode ->                  t -> t   ->       t*t
val gquo        : round_mode ->                  t -> t   ->       t
val gmod        : round_mode ->                  t -> t   ->       t

val quomod_in   :              tref -> tref ->   t -> t   ->       unit
val quo_in      :              tref ->           t -> t   ->       unit
val mod_in      :                      tref ->   t -> t   ->       unit
val gquomod_in  : round_mode -> tref -> tref ->  t -> t   ->       unit
val gquo_in     : round_mode -> tref ->          t -> t   ->       unit
val gmod_in     : round_mode ->         tref ->  t -> t   ->       unit

val quomod_1    :                                t -> int ->       t*int
val quo_1       :                                t -> int ->       t
val mod_1       :                                t -> int ->       int
val gquomod_1   : round_mode ->                  t -> int ->       t*int
val gquo_1      : round_mode ->                  t -> int ->       t
val gmod_1      : round_mode ->                  t -> int ->       int

val quomod_1_in :              tref ->           t -> int ->       int
val quo_1_in    :              tref ->           t -> int ->       unit
val gquomod_1_in: round_mode -> tref ->          t -> int ->       int
val gquo_1_in   : round_mode -> tref ->          t -> int ->       unit
```

```
(* valeur absolue ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val abs         :                                t ->              t
val abs_in      :               tref ->          t ->              unit

(* opposé         ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val neg         :                                t ->              t
val neg_in      :               tref ->          t ->              unit

(* puissance p-ème ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val sqr         :                                t ->              t
val pow         :                                t -> int ->       t
val pow_1       :                              int -> int ->       t
val powmod      :                                t -> t   -> t ->  t
val gpowmod     : round_mode ->                  t -> t   -> t ->  t
val sqr_in      :               tref ->          t ->              unit
val pow_in      :               tref ->          t -> int ->       unit
val pow_1_in    :               tref ->        int -> int ->       unit
val powmod_in   :               tref ->          t -> t   -> t ->  unit
val gpowmod_in  : round_mode -> tref ->          t -> t   -> t ->  unit

(* racine p-ème   ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val sqrt        :                                t ->              t
val root        :                                t -> int ->       t
val gsqrt       : round_mode ->                  t ->              t
val groot       : round_mode ->                  t -> int ->       t
val sqrt_in     :               tref ->          t ->              unit
val root_in     :               tref ->          t -> int ->       unit
val gsqrt_in    : round_mode -> tref ->          t ->              unit
val groot_in    : round_mode -> tref ->          t -> int ->       unit

(* factorielle    ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
val fact        :                              int ->              t
val fact_in     :               tref ->        int ->              unit

(* pgcd           ------------------------------------------------- *)
(*                  d     u     v     p    q    a    b      c      res  *)
val gcd         :                                t -> t   ->       t
val gcd_ex      :                                t -> t   ->       t*t*t
val cfrac       :                                t -> t   ->    t*t*t*t*t
val gcd_in      : tref->                         t -> t   ->       unit
val gcd_ex_in   : tref->tref->tref->             t -> t   ->       unit
val cfrac_in    : tref->tref->tref->tref->tref->t -> t   ->       unit

(* comparaison    ------------------------------------------------- *)
(*                mode          r       s       a    b      c      res  *)
```

```
val sgn         :                              t ->              int
val cmp         :                              t -> t   ->       int
val cmp_1       :                              t -> int ->       int
val eq          :                              t -> t   ->       bool
val eq_1        :                              t -> int ->       bool
val neq         :                              t -> t   ->       bool
val neq_1       :                              t -> int ->       bool
val inf         :                              t -> t   ->       bool
val inf_1       :                              t -> int ->       bool
val infeq       :                              t -> t   ->       bool
val infeq_1     :                              t -> int ->       bool
val sup         :                              t -> t   ->       bool
val sup_1       :                              t -> int ->       bool
val supeq       :                              t -> t   ->       bool
val supeq_1     :                              t -> int ->       bool

(* conversion     ------------------------------------------------------ *)
(*               mode          r       s       a    b      c      res  *)
val of_int      :                              int ->            t
val of_string   :                           string ->            t
val of_int_in   :              tref ->         int ->            unit
val of_string_in:              tref ->      string ->            unit
val int_of      :                              t ->              int
val string_of   :                              t ->              string
val bstring_of  :                              t ->              string
val hstring_of  :                              t ->              string
val ostring_of  :                              t ->              string

(* nombre aléatoire ----------------------------------------------------- *)
(*               mode          r       s       a    b      c      res  *)
val nrandom     :                             int->              t
val zrandom     :                             int->              t
val nrandom1    :                             int->              t
val zrandom1    :                             int->              t
val nrandom_in  :              tref ->        int->              unit
val zrandom_in  :              tref ->        int->              unit
val nrandom1_in :              tref ->        int->              unit
val zrandom1_in :              tref ->        int->              unit
val random_init :                             int->              unit

(* représentation binaire ----------------------------------------------- *)
(*               mode          r       s       a    b      c      res  *)
val nbits       :                              t ->              int
val lowbits     :                              t ->              int
val highbits    :                              t ->              int
val nth_word    :                              t -> int ->       int
val nth_bit     :                              t -> int ->       bool

(* décalage       ------------------------------------------------------ *)
(*               mode          r       s       a    b      c      res  *)
```

```
  val shl         :                              t -> int ->        t
  val shr         :                              t -> int ->        t
  val split       :                              t -> int ->        t*t
  val join        :                              t -> t   -> int -> t
  val shl_in      :              tref ->         t -> int ->        unit
  val shr_in      :              tref ->         t -> int ->        unit
  val split_in    :              tref -> tref -> t -> int ->        unit
  val join_in     :              tref ->         t -> t   -> int -> unit

  (* affichage      ------------------------------------------------------ *)
  (*                mode           r       s       a    b      c      res  *)
  val toplevel_print     :                         t ->               unit
  val toplevel_print_tref:        tref ->                             unit

  (* exceptions *)
  exception Error of string

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      (* div      *)
  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 (  +=  ) : tref -> t -> unit    (* add_in   *)
  val (  -=  ) : tref -> t -> unit    (* sub_in   *)
  val (  *=  ) : tref -> t -> unit    (* mul_in   *)
  val (  /=  ) : tref -> t -> unit    (* quo_in   *)
  val (  %=  ) : tref -> t -> unit    (* mod_in   *)

  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 -> t      (* pow_1    *)
```

```
  val (  +=. ) : tref -> int -> unit  (* add_1_in *)
  val (  -=. ) : tref -> int -> unit  (* sub_1_in *)
  val (  *=. ) : tref -> int -> unit  (* mul_1_in *)
  val (  /=. ) : tref -> int -> unit  (* quo_1_in *)

  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 (  ~~  ) : tref -> t            (* look     *)

end (* foncteur Infixes *)

                        (* +------------------------+
                           |  Modules disponibles   |
                           +------------------------+ *)

(* Les modules suivants implémentent tous la même signature Int_type.
   Certains modules peuvent être indisponibles sur une machine
   particulière si son architecture matérielle ou logicielle ne permet
   pas leur compilation.                                              *)

module Big   : Int_type
module Clong : sig ... end   (* descriptions concrètes *)
module Dlong : sig ... end   (* compatibles avec la    *)
module Slong : sig ... end   (* signature Int_type     *)
module Gmp   : sig ... end

(* comparaison entre deux modules *)
module Cmp(A:Int_type)(B:Int_type) : Int_type

(* statistiques *)
module Count(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                           *)
  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 (* affichage     *)

  include Int_type

end (* foncteur Count *)

     (* +------------------------------------------------------------+
        |  Approximation des fonctions à valeurs réelles usuelles  |
        +------------------------------------------------------------+ *)

module Rfuns(E:Int_type) : sig

  exception Error of string

  (* [f a b n] retourne x tel que |2^n*f(a/b) - x| < 1 *)
  val arccos    : E.t -> E.t -> int -> E.t
  val arccosh   : E.t -> E.t -> int -> E.t
  val arccot    : E.t -> E.t -> int -> E.t
  val arccoth   : E.t -> E.t -> int -> E.t
  val arcsin    : E.t -> E.t -> int -> E.t
  val arcsinh   : E.t -> E.t -> int -> E.t
  val arctan    : E.t -> E.t -> int -> E.t
  val arctanh   : E.t -> E.t -> int -> E.t
  val arg       : E.t -> E.t -> int -> E.t
  val cos       : E.t -> E.t -> int -> E.t
  val cosh      : E.t -> E.t -> int -> E.t
  val cosin     : E.t -> E.t -> int -> E.t*E.t
  val cosinh    : E.t -> E.t -> int -> E.t*E.t
  val cot       : E.t -> E.t -> int -> E.t
  val coth      : E.t -> E.t -> int -> E.t
  val exp       : E.t -> E.t -> int -> E.t
  val ln        : E.t -> E.t -> int -> E.t
  val sin       : E.t -> E.t -> int -> E.t
  val sinh      : E.t -> E.t -> int -> E.t
  val tan       : E.t -> E.t -> int -> E.t
  val tanh      : E.t -> E.t -> int -> E.t

  (* [r_f r a b c] retourne l'entier approchant c*f(a/b) selon
                   le mode d'arrondi r *)
  val r_arccos  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccosh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccot  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arccoth : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arcsin  : round_mode -> E.t -> E.t -> E.t -> E.t
```

```
  val r_arcsinh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arctan  : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arctanh : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_arg     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cos     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cosh    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_cosin   : round_mode -> E.t -> E.t -> E.t -> E.t*E.t
  val r_cosinh  : round_mode -> E.t -> E.t -> E.t -> E.t*E.t
  val r_cot     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_coth    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_exp     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_ln      : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_sin     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_sinh    : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_tan     : round_mode -> E.t -> E.t -> E.t -> E.t
  val r_tanh    : round_mode -> E.t -> E.t -> E.t -> E.t

  (* création d'une r_fonction *)
  val round : (int -> E.t) -> round_mode -> E.t -> E.t

  (* gestion du cache *)
  val cache_bits  : unit -> int
  val clear_cache : unit -> unit

end (* foncteur Rfuns *)

                    (* +---------------------------+
                       |  Sélection à l'exécution  |
                       +---------------------------+ *)

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

                         (* +-----------------+
                            |  Chronométrage  |
                            +-----------------+ *)

external chrono : string -> unit = "chrono"
```

2.2 La signature `Int_type`

2.2.1 Entiers et références

Une implémentation de la signature `Int_type` fournit deux types de données :

- Le type `t` représente un nombre entier relatif. La longueur en bits d'un tel entier n'est limitée que par la quantité de mémoire disponible sur la machine, et dans le cas des modules `Clong`, `Dlong`, `Slong` et `Big` par la quantité de mémoire maximale qui peut être allouée à une donnée Ocaml (2^{27} bits sur une machine 32 bits, 2^{60} bits sur une machine 64 bits).

- Le type `tref` représente une zone mémoire mutable et extensible pouvant recevoir une donnée de type `t`. Cette zone mémoire est étendue selon une politique de doublement de la taille lorsque sa capacité courante est insuffisante pour la donnée qui doit y être stockée. Une zone mémoire de type `tref` n'est jamais rétrécie.

On crée une référence de type `tref` grâce à la fonction `make_ref` qui effectue une copie physique de son argument et retourne le pointeur sur la zone mémoire allouée à cette copie. On stocke un nouvel entier dans une référence de type `tref` grâce aux fonctions `xxx_in` qui effectuent le calcul désigné par `xxx` et stockent le résultat dans l'argument de type `tref` transmis à `xxx_in`. Lorsqu'une fonction `xxx` calcule plusieurs résultats de type `t`, la fonction `xxx_in` associée prend en arguments supplémentaires autant de références qu'il y a de résultats à stocker ; ces arguments doivent désigner des emplacements mémoire distincts.

On peut récupérer l'entier de type `t` stocké dans une référence de type `tref` grâce aux fonctions `copy_out` et `look` :

- `copy_out` effectue une copie physique de l'entier à récupérer et retourne un pointeur vers cette copie. Toute action ultérieure sur la référence est sans effet sur la copie obtenue.

- `look` n'effectue aucune copie et retourne un pointeur vers la zone mémoire associée à la référence. L'entier obtenu par `look r` est ainsi volatile, c'est-à-dire que sa valeur peut changer à la suite du stockage d'un nouvel entier dans `r` (elle peut aussi ne pas changer si ce stockage provoque l'extension de la zone mémoire référencée).

Il est recommandé de n'utiliser `look` que dans les calculs intermédiaires où l'on souhaite éviter une copie pour des raisons de performance. Les opérations de type lecture-modification-écriture, par exemple `xxx_in r (look r) z`, sont traitées correctement.

2.2.2 Différentes versions d'une même opération

En général une opération entre grands entiers est fournie en quatre versions :

- `xxx : t -> t -> t` : calcule un résultat de type `t` à partir de deux opérandes de type `t`.

– `xxx_1 : t -> int -> t` : calcule un résultat de type `t` à partir d'un opérande de type `t` et d'un opérande de type `int`. `xxx_1` a b est formellement équivalente à `xxx_1` a (`of_int` b), mais est en général implémentée plus efficacement, de façon à économiser le temps nécessaire à l'allocation d'un résultat intermédiaire et à limiter la charge de travail du gestionnaire mémoire intégré à Ocaml lors de la récupération des zones mémoires inutilisées.

– `xxx_in : tref -> t -> t -> unit` : calcule un résultat de type `t` à partir de deux opérandes de type `t`, et stocke ce résultat dans la zone mémoire désignée par la référence de type `tref`. `xxx_in` r a b est formellement équivalente à `copy_in` r (`xxx` a b), mais en général le résultat est calculé directement dans la zone mémoire désignée par r, ce qui économise le temps nécessaire à l'allocation du résultat, à sa copie, et le temps de récupération de la mémoire temporaire.

– `xxx_1_in : tref -> t -> int -> unit` : calcule un résultat de type `t` à partir de deux opérandes de type `t` et `int`, et stocke ce résultat dans la zone mémoire désignée par l'opérande de type `tref`. `xxx_1_in` r a b est formellement équivalente à `copy_in` r (`xxx` a (`of_int` b)), avec les mêmes économies en termes de charge du gestionnaire mémoire que pour `xxx_1` et `xxx_in`.

2.2.3 Mode d'arrondi

Les opérations produisant une approximation entière d'un nombre réel a (division, racine carrée et racine p-ème) sont fournies en cinq versions :

`xxx`	*args*	calcule $\lfloor a \rfloor$
`gxxx Floor`	*args*	calcule $\lfloor a \rfloor$
`gxxx Ceil`	*args*	calcule $\lceil a \rceil$
`gxxx Nearest_up`	*args*	calcule $\lfloor a + 1/2 \rfloor$
`gxxx Nearest_down`	*args*	calcule $\lceil a - 1/2 \rceil$

Noter que les modes d'arrondi `Nearest_up` et `Nearest_down` ne produisent un résultat différent que si $a = k + \frac{1}{2}$ avec k entier : `Nearest_up` retourne $k+1$ tandis que `Nearest_down` retourne k.

2.2.4 Opérations arithmétiques

Le tableau ci-dessous donne les descriptions mathématiques des opérations arithmétiques implémentées dans un module de signature `Int_type`. Les lettres a, b, c désignent des valeurs de type `t` ou `int`, la lettre n désigne un opérande de type `int`. Lorsqu'une opération `xxx` retourne plusieurs résultats, l'opération `xxx_in` associée place les résultats dans les références transmises en argument dans le même ordre.

opération	arguments	résultats
`add`	a b	$a+b$
`sub`	a b	$a-b$
`mul`	a b	ab
`quomod`	a b	$(\lfloor a/b \rfloor, a - \lfloor a/b \rfloor b)$
`quo`	a b	$\lfloor a/b \rfloor$
`modulo`	a b	$a - \lfloor a/b \rfloor b$
`abs`	a	$\lvert a \rvert$
`neg`	a	$-a$
`sqr`	a	a^2
`pow`	a n	a^n
`powmod`	a b c	$a^b - \lfloor a^b/c \rfloor c$
`sqrt`	a	$\lfloor \sqrt{a} \rfloor$
`root`	a n	$\lfloor \sqrt[n]{a} \rfloor$
`fact`	n	$n!$
`gcd`	a b	d
`gcd_ex`	a b	(d,u,v)
`cfrac`	a b	(d,u,v,p,q)

`cfrac` a b retourne un quintuplet (d,u,v,p,q) tel que d est le pgcd positif de a et b, $ua - vb = d$, $pu - qv = 1$, $pa = qb$ 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'une implémentation de la signature `Int_type` à l'autre. `gcd_ex` a b retourne le triplet (d,u,v) et `gcd` a b retourne d. Noter que l'ordre des résultats a été inversé par rapport à `Numerix`-0.19, par souci de cohérence avec les interfaces C et Pascal.

2.2.5 Comparaisons

`sgn` a retourne 1 si $a > 0$, 0 si $a = 0$ et -1 si $a < 0$. `cmp` a b est formellement équivalent à `sgn`$(a - b)$, mais la soustraction n'est pas réellement effectuée : a et b sont comparés bit à bit en commençant par les bits de poids fort jusqu'à ce que le signe de la différence soit déterminé.

Les opérations de comparaison à résultat booléen sont accessibles avec les identificateurs indiqués dans la signature `Int_type`. Les modules `Clong`, `Dlong`, `Slong` et `GMP` permettent aussi de comparer deux valeurs de type `t` avec les symboles de comparaison infixes polymorphes de Ocaml selon la correspondance suivante :

```
eq    =      inf     <      sup     >
neq   <>     infeq   <=     supeq   >=
```

Le module `Big` n'implémente pas les opérations de comparaison polymorphes et donc seuls les identificateurs préfixes `eq`, ..., `supeq` sont utilisables avec ce module. Noter que le correctif optionnel inclus dans `Numerix`-0.19 pour palier cette déficience du module `Big` n'a pas été repris dans `Numerix`-0.21. En effet, ce correctif fournissait des résultats faux lors de la comparaison d'entiers négatifs et il n'y a pas moyen avec l'implémentation actuelle de `Big_int` de contourner ce problème.

2.2.6 Conversions

`of_int` convertit une valeur de type `int` en la valeur correspondante de type `t`. `int_of` effectue la conversion inverse sous réserve que l'entier à convertir ait une valeur absolue strictement inférieure à 2^{30}, sinon une exception est déclenchée. Noter que le seuil 2^{30} est indépendant de l'architecture 32 ou 64 bits de la machine.

`of_string` s retourne l'entier de type `t` représenté par la chaîne s conformément à la syntaxe suivante :

- Un signe `+` ou `-` optionnel en tête.
- Un préfixe `0x`, `0X`, `0o`, `0O`, `0b` ou `0B` après le signe optionnel, indiquant la base de numération 16, 8 ou 2. La base 10 est indiquée par l'absence de préfixe.
- Une suite non vide de chiffres, valides pour la base de numération, sans espace ni caractère de soulignement. Pour la base 16 les lettres `a,b,c,d,e,f` et `A,B,C,D,E,F` sont acceptées.

La conversion d'une valeur de type `t` en chaîne de caractères est effectuée avec l'une des fonctions suivantes : `string_of` (base 10), `hstring_of` (base 16), `ostring_of` (base 8), `bstring_of` (base 2). La chaîne produite est conforme à la syntaxe acceptée par `of_string`, ce qui permet de convertir une valeur a de type `A.t` en la valeur correspondante b de type `B.t`, `A` et `B` désignant deux implémentations compatibles avec la signature `Int_type`, avec l'instruction :

`let` b `= B.of_string(A.hstring_of` a`)`

Il est recommandé d'utiliser la conversion en base 16 pour ce faire, car c'est celle fournissant la chaîne la plus compacte et elle a une complexité linéaire en la taille en bits de a. Noter que cette méthode de conversion ne fonctionne pas si la taille de la représentation hexadécimale de a est supérieure à la taille maximale d'une chaîne de caractères autorisée par Ocaml (soit $|a| \geq 16^{2^{24}-4}$ sur une machine 32 bits et $|a| \geq 16^{2^{57}-4}$ sur une machine 64 bits). Dans ce cas, `A.hstring_of` a retourne la chaîne `"<very long number>"` qui sera rejetée par `B.of_string`.

Les fonctions `toplevel_print` et `toplevel_print_tref` convertissent une valeur de type `t` ou `tref` en sa représentation décimale et l'affichent à l'aide des fonctions d'impression du module `Format`. Lorsque la chaîne à afficher comporte plus de 1000 caractères, il est seulement affiché les 200 premiers caractères, le nombre de caractères supprimés, puis les 200 derniers.

2.2.7 Nombres pseudo-aléatoires

Les fonctions `nrandom`, `nrandom1`, `zrandom` et `zrandom1` retournent des entiers pseudo-aléatoires ayant n bits où n est un nombre positif passé en argument. Pour `nrandom` et `nrandom1` le résultat est compris entre 0 et $2^n - 1$, pour `zrandom` et `zrandom1` le résultat est compris entre $-2^n + 1$ et $2^n - 1$. Pour

`nrandom1` et `zrandom1` le n-ème bit du résultat vaut 1, c'est-à-dire que sa valeur absolue est supérieure ou égale à 2^{n-1}. Noter que la fonction `random_bits` de `Numerix`-0.19 a été remplacée par `nrandom` dans `Numerix`-0.21.

Le générateur pseudo-aléatoire utilisé dépend du module et de la machine utilisés, donc le comportement d'un programme utilisant ces fonctions est à priori non reproductible si l'on change de module ou de machine. La fonction `random_init` permet d'initialiser à la fois le générateur pseudo-aléatoire du module et celui de Ocaml à partir d'une graine de type `int`. Si cette graine est nulle, elle est remplacée par la date en secondes à laquelle l'instruction `random_init` est exécutée. La séquence obtenue à partir d'une graine non nulle est reproductible, pour une combinaison (module,machine) donnée, en réinitialisant le générateur avec la même graine.

2.2.8 Accès à la représentation binaire

Si a et b sont des valeurs de type `t` alors :

- `nbits` a retourne le nombre de bits de $|a|$, soit 0 pour $a = 0$ et $\lceil \log_2 |a| \rceil$ pour $a \neq 0$.

- `lowbits` a retourne les 31 bits de poids faible de $|a|$, soit $|a| \bmod 2^{31}$.

- `highbits` a retourne les 31 bits de poids fort de $|a|$, soit $\lfloor |a|/2^{31-\texttt{nbits}(a)} \rfloor$. Noter que pour $a \neq 0$ le nombre ainsi obtenu est considéré comme négatif par Ocaml sur une machine 32 bits.

- `nth_word` a retourne le nombre constitué des bits de $|a|$ de rang $16n$ à $16n+15$, soit $\lfloor |a|/2^{16n} \rfloor \bmod 2^{16}$. Si $n < 0$ ou $n > \texttt{nbits}(a)/16$, le résultat est nul.

- `nth_bit` a retourne le n-ème bit de $|a|$, soit `true` si $\lfloor |a|/2^n \rfloor$ est impair, et `false` sinon. Si $n < 0$ ou $n > \texttt{nbits}(a)$, le résultat est `false`.

- `shl` a n retourne le nombre ayant même signe que a et obtenu par décalage de $|a|$ de n bits vers la gauche si $n \geq 0$ ou $-n$ bits vers la droite si $n < 0$, soit $\texttt{sgn}(a)\lfloor 2^n |a| \rfloor$ dans les deux cas.

- `shr` a n retourne le nombre ayant même signe que a et obtenu par décalage de $|a|$ de n bits vers la droite si $n \geq 0$ ou $-n$ bits vers la gauche si $n < 0$, soit $\texttt{sgn}(a)\lfloor |a|/2^n \rfloor$ dans les deux cas.

- `split` a n retourne le couple (q, r) tel que $|q| = \lfloor |a|/2^n \rfloor$, $|r| = |a| \bmod 2^n$, $qa \geq 0$ et $ra \geq 0$. n doit être positif ou nul.

- `join` a b n retourne le nombre $a + 2^n b$, n doit être positif ou nul.

2.2.9 Hachage, sérialisation et désérialisation

Les modules `Clong`, `Dlong`, `Slong` et `GMP` comportent des interfaces avec la fonction de hachage générique de Ocaml. La clé de hachage d'un grand entier de l'un de ces modules est calculée à partir de sa représentation binaire, donc peut varier selon le module utilisé. Le module `Big_int` comporte une interface minimale avec la fonction de hachage générique : seul le signe d'un nombre est pris en compte pour former la clé de hachage. De ce fait, les grands entiers de ces cinq modules peuvent être inclus dans des tables de hachage utilisant la fonction `Hashtbl.hash`, avec toutefois un taux de collisions élevé en ce qui concerne le module `Big`.

Par ailleurs, tous les modules comportent des interfaces avec les fonctions de sérialisation et de désérialisation de Ocaml, donc les grands entiers peuvent être exportés ou importés via les fonctions `output_value` et `input_value` et peuvent être encodés en suites d'octets ou décodés à partir de telles suites via les fonctions du module `Marshal`. Noter que le typage doit être préservé entre l'exportation ou l'encodage et l'importation ou le décodage d'un grand entier, 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.

2.2.10 Erreurs

Les modules `Clong`, `Dlong`, `Slong` et `GMP` contrôlent la validité des arguments des fonctions qu'ils implémentent et déclenchent en cas d'argument invalide l'une des exceptions `Error` *msg* suivantes :

msg	cause
`integer overflow`	`int_of` a avec $\|a\| \geq 2^{30}$
`invalid string`	`of_string` avec une chaîne invalide
`multiple result`	`xxx_in` avec plusieurs références identiques
`negative base`	`fact` n avec $n < 0$, `sqrt` a avec $a < 0$, `root` a n avec $a < 0$ et n pair
`negative exponent`	`pow` et `powmod` quand l'exposant est négatif `root` quand l'exposant est négatif ou nul
`negative index`	`split`, `join` avec $n < 0$
`negative size`	`xrandom`, `xrandom1` avec $n < 0$
`number too big`	résultat trop grand pour tenir dans une valeur Ocaml
`division by zero`	`quoxxx`, `modxxx`, `powmod` quand le diviseur est nul

En ce qui concerne le module `Big`, un argument invalide peut déclencher une exception soit au niveau du code d'interfaçage à `Numerix`, soit au niveau du module `Big_int` de Ocaml. Dans le premier cas, l'exception déclenchée est conforme au tableau ci-dessus ; dans le second cas, il est déclenché une exception spécifique à `Big_int`, et non conforme à ce tableau.

Par ailleurs, le noyau C de `Numerix` (qui implémente les modules `Clong`, `Dlong` et `Slong`) peut déclencher l'une des erreurs non rattrapables suivantes :

`"Numerix kernel: out of memory"` : un calcul ne peut pas être conduit à terme car il n'y a pas assez de mémoire disponible.

`"Numerix kernel: number too big"` : un calcul ne peut pas être conduit à terme car il nécessite un résultat intermédiaire trop grand.

`"Numerix kernel: xxx"` : le code C a détecté un bogue interne à `Numerix`. Ceci ne doit pas arriver dans la version utilisateur de `Numerix` car les instructions de contrôle interne sont désactivées par défaut. Si vous rencontrez une telle erreur, merci de me la signaler.

2.3 Les foncteurs utilisant la signature `Int_type`

2.3.1 Symboles infixes

Le foncteur `Infixes` prend en argument un module conforme à la signature `Int_type` et définit des équivalents infixes pour les opérations les plus courantes de ce module. Les opérations infixes entre une référence de type `tref` et une valeur de type `t` ou de type `int` sont conformes à la syntaxe de C, par exemple `r -= a` est interprété comme `sub_in r (look r) a`. Noter que le foncteur `Infixes` de `Numerix`-0.21 est incompatible avec celui de `Numerix`-0.19 : ce dernier surchargeait les opérateurs usuels entre valeurs de type `int`, ce qui rendait son emploi malaisé.

2.3.2 Comparaison entre deux modules

Le foncteur `Cmp` prend en arguments deux modules `A` et `B` conformes à la signature `Int_type` et retourne un module `C` conforme à cette signature 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 indiquant sous forme textuelle la fonction en cause et les arguments et résultats dans chaque module. 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 `A` sont convertis en valeurs de type `B.t` pour former des résultats de type `C.t`. De même, le générateur pseudo-aléatoire de `C` est construit à partir de celui de `A` seul.

2.3.3 Statistiques

Le foncteur `Count` prend en argument un module `A` conforme à la signature `Int_type` et retourne un module `B` conforme à cette signature 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 :

`cadd`	compte les additions et soustractions ;
`cmul`	compte les multiplications et les élévations au carré ;
`cquo`	compte les divisions ;
`cpow`	compte les exponentiations et les factorielles ;
`croot`	compte les racines carrées et les racines p-èmes ;
`cgcd`	compte les pgcd et les opérations associées ;
`cbin`	compte les opérations sur les représentations binaires ;
`cmisc`	compte toutes les autres opérations à l'exception de `look`, `random_init`, `toplevel_print` et `toplevel_print_tref`.

Chaque compteur `cxxx` comporte trois champs :

`cxxx.n`	nombres d'appels à l'une des fonctions associées à `cxxx` ;
`cxxx.s`	somme des tailles en bits des arguments ;
`cxxx.m`	maximum des tailles en bits des arguments.

Pour chaque appel à une fonction de `B` 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.

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 `clear_stats` remet à zéro tous les compteurs et la fonction `print_stats` affiche les statistiques associées à chaque compteur (nombre d'opérations, taille moyenne et taille maximum d'un opérande).

2.3.4 Approximations des fonctions usuelles

Le foncteur `Rfuns` prend en argument un module `E` conforme à la signature `Int_type` et retourne un module implémentant des algorithmes d'approximation pour les fonctions mathématiques usuelles :

arccos	arccosh	arccot	arccoth	arcsin	arcsinh
arctan	arctanh	cos	cosh	cot	coth
exp	ln	sin	sinh	tan	tanh

A l'exception de arccot, toutes les fonctions ci-dessus sont réputées être définies mathématiquement de manière non ambiguë. La fonction arcot implémentée dans `Numerix` est définie mathématiquement par :

$$(\operatorname{arccot} x = \theta) \iff (\cot\theta = x \text{ et } 0 < \theta < \pi).$$

La plupart des logiciels mathématiques implémentent une définition différente en imposant $-\pi/2 < \theta < \pi/2$, mais cela introduit une discontinuité artificielle en 0 et je considère ma propre définition comme meilleure.

f désignant l'une des fonctions précédentes, deux interfaces à l'algorithme d'approximation de f sont disponibles :

```
f   :                E.t -> E.t -> int -> E.t
r_f : round_mode -> E.t -> E.t -> E.t -> E.t
```

`f` a b n retourne un entier x tel que $x - 1 < 2^n f(a/b) < x + 1$, c'est-à-dire l'un des deux nombres $x_1 = \lfloor 2^n f(a/b) \rfloor$, $x_2 = \lceil 2^n f(a/b) \rceil$. Les valeurs négatives pour n sont acceptées. Les entiers a et b ne doivent pas être tous deux nuls, et a/b doit appartenir au domaine de définition de f. Le quotient $a/0$ est considéré comme égal à $+\infty$ ou $-\infty$ selon le signe de a, il est accepté si f admet une limite finie en ce point. Lorsque $x_1 \neq x_2$, il est à priori impossible de prédire quelle valeur sera retournée : cela dépend des nombres a et b, ainsi que de l'état du cache utilisé par l'algorithme d'approximation de f.

`r_f Floor`	a b c retourne $\lfloor cf(a/b) \rfloor$,
`r_f Ceil`	a b c retourne $\lceil cf(a/b) \rceil$,
`r_f Nearest_up`	a b c retourne $\lfloor cf(a/b) + 1/2 \rfloor$,
`r_f Nearest_down`	a b c retourne $\lceil cf(a/b) - 1/2 \rceil$.

Les paramètres a et b sont soumis aux mêmes contraintes que pour `f`. La valeur retournée étant définie de manière unique, elle est indépendante de l'état du cache utilisé par l'algorithme d'approximation de f. Noter que les modes d'arrondi `Nearest_up` et `Nearest_down` sont équivalents pour les fonctions f disponibles car $cf(a/b)$ ne peut pas être de la forme $k + \frac{1}{2}$ avec k entier pour ces fonctions.

D'un point de vue performances, il est recommandé d'utiliser la première interface (fonction `f`), car à l'exception des fonctions `cot` et `tan`, le calcul de `f` a b n a une complexité $O(M(k) \ln k)$ où $M(k)$ désigne la complexité d'une multiplication de deux entiers dont le produit tient sur k bits et

$$k = \max(\texttt{nbits}(a), \texttt{nbits}(b), \texttt{nbits}(\lfloor 2^n f(a/b) \rfloor)),$$

tandis que le calcul de `r_f` r a b c a une complexité non bornée (l'algorithme implémentant `r_f` consiste à calculer `f` a b n avec des valeurs de plus en plus grandes pour n jusqu'à obtenir un résultat permettant de déterminer l'arrondi correct de $cf(a/b)$). Les complexités de `cot` et `tan` sont non bornées pour la même raison : on peut avoir à calculer des valeurs arbitrairement précises de $\cos(a/b)$ et $\sin(a/b)$ lorsque a/b est proche d'un multiple de $\pi/2$.

Les fonctions suivantes sont également disponibles avec les deux interfaces :

arg :	$(\arg(x, y) = \theta) \iff (x + iy = e^{i\theta}\sqrt{x^2 + y^2}$ et $-\pi < \theta \leq \pi)$.
cosin :	$\operatorname{cosin} x = (\cos x, \sin x)$,
cosinh :	$\operatorname{cosinh} x = (\cosh x, \sinh x)$,

Formellement, `cosin` a b n retourne le couple (`cos` a b n, `sin` a b n), et il est recommandé d'utiliser la fonction `cosin` plutôt que d'appeler séparément `cos` et `sin` lorsque l'on veut des approximations du cosinus et du sinus d'un même

angle. Des considérations analogues s'appliquent aux fonctions `r_cosin`, `cosinh` et `r_cosinh`.

En ce qui concerne les fonctions `arg` et `r_arg`, leur usage est préférable à celui de `arccos`, `arcsin` `r_arccos` et `r_arcsin` car ces quatre dernières fonctions sont implémentées par appel à `arg` ou `r_arg` après un calcul de racine carrée potentiellement coûteux. Les fonctions `arctan`, `r_arctan`, `arccot` et `r_arccot` appellent elles aussi `arg` ou `r_arg`, mais sans effectuer de calcul préalable coûteux donc leur usage n'est pas déconseillé.

Le mécanisme d'accroissement progressif de précision implémenté dans les fonctions `r_xxx` est disponible pour l'utilisateur grâce à la fonction `round` : soit t un nombre réel irrationnel et `f : int -> E.t` une fonction d'approximation de t telle que pour tout n entier `f` n retourne un entier x vérifiant $x-1 < 2^n t < x+1$. Alors `round f` retourne une fonction `r_f : round_mode -> E.t -> E.t` telle que `r_f` r c retourne l'arrondi selon le mode indiqué par r du réel ct. Noter que le calcul de `r_f` r c ne peut pas boucler, même si t est rationnel. Dans le pire des cas, ce calcul terminera sur une erreur pour mémoire insuffisante ou pour nombre trop grand.

Les algorithmes d'approximation implémentés dans le foncteur `Rfuns` font usage d'une mémoire cache où sont stockées les approximations de certaines constantes fréquemment utilisées. Si l'une de ces approximations se révèle insuffisamment précise pour le calcul en cours, alors une nouvelle approximation est calculée avec une précision suffisante pour pouvoir terminer le calcul en cours et cette nouvelle approximation remplace l'ancienne dans la mémoire cache. Les constantes stockées dans la mémoire cache ont été choisies de façon à pouvoir obtenir pour un coût minime (quelques additions et un décalage) les approximations des nombres suivants :

$\ln(2)$	$\exp(1)$	$\arctan(1)$
$\ln(3)$	$\exp(-1)$	$\arctan(1/2)$
$\ln(5)$	$\cos(1)$	$\arctan(1/3)$
	$\sin(1)$	$\arctan(1/5)$

La fonction `cache_bits` retourne la somme des tailles en bits des approximations présentes dans la mémoire cache ; la taille totale de la mémoire cache est environ le double de la valeur retournée par `cache_bits`. La fonction `clear_cache` restaure les approximations initiales sur 100 bits des constantes cachées, permettant ainsi au gestionnaire mémoire de Ocaml de récupérer la mémoire occupée par le cache.

L'utilisation de cette mémoire cache permet d'accélérer les calculs demandés par l'utilisateur, mais elle a pour effet secondaire de rendre non reproductible un calcul quelconque de la forme `f` a b n : la valeur retournée peut varier selon la précision avec laquelle sont connues les constantes utilisées par `f`. Cependant le mécanisme de gestion du cache permet de garantir une cohérence avec le passé : si un calcul `f` a b n a retourné une valeur x alors tout calcul ultérieur avec les mêmes arguments retournera la même valeur x, même si la précision des constantes cachées a été améliorée entre-temps. Naturellement la garantie

de cohérence avec le passé prend fin si l'on réinitialise la mémoire cache avec la fonction `clear_cache`.

Les fonctions du module `Rfuns(E)` peuvent déclencher en cas de problème les exceptions `Rfuns(E).Error` *msg* suivantes :

msg	cause
`0/0`	$a = b = 0$
`number too big`	voir ci-dessous

`arcsinh`
`cos`
`cosin`
`sin`
`tan`
	$a/b = \pm\infty$

`arccos` `arcsin`	$\|a/b\| > 1$
`arccosh`	$a/b = +\infty$ ou $a/b < 1$
`arctanh`	$\|a/b\| \geq 1$
`cot`	$a/b = 0$ ou $a/b = \pm\infty$
`coth`	$a/b = 0$
`exp`	$a/b = +\infty$
`ln`	$a/b \leq 0$ ou $a/b = +\infty$

Dans les cas où le calcul d'un résultat intermédiaire est impossible parce que sa taille est trop grande, il est déclenché l'une des exceptions suivantes :

- `Rfuns(E).Error "number too big"` : le dépassement de taille a été détecté par une fonction de `Rfuns`.
- `E.Error "number too big"` : le dépassement de taille a été détecté par une fonction de `E`.
- `"Numerix kernel: number too big"` : le dépassement de taille a été détecté par le noyau C de `Numerix`. Dans ce cas, l'exception est non rattrapable.

2.3.5 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 conforme à la signature `Int_type` et fournissant une implémentation de la fonction `main : string list -> unit` qui constitue le point d'entrée du programme.

`Start(Main).start` examine la ligne de commande, sélectionne un module `E` conforme à la signature `Int_type` en fonction des options `-e xxx` et `-count` qui y figurent, puis appelle `Main(E).main` avec en argument la liste des autres paramètres de la ligne de commande, y compris le paramètre de rang zéro qui

désigne généralement le nom du programme (ce paramètre n'était pas transmis dans `Numerix`-0.19).

L'option `-e xxx` sélectionne un module parmi `Clong`, `Dlong`, `Slong`, `GMP`, `Big` où `xxx` est le nom de ce module en minuscules. Si plusieurs options `-e xxx` sont présentes sur la ligne de commande, seules les deux dernières sont conservées et ces deux dernières options sélectionnent le module `Cmp(A)(B)`, `A` étant le module désigné par l'avant-dernière option et `B` celui désigné par la dernière. Si aucune option `-e xxx` n'est présente, le module sélectionné est le premier module disponible dans l'ordre `Clong`, `Dlong`, `Slong`, `GMP`, `Big`.

L'option `-count` sélectionne le module `Count(E)` où `E` est le module sélectionné par les options `-e xxx`.

2.3.6 Chronométrage

`chrono` *msg* affiche sur le canal de sortie 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. Noter que `Numerix`-0.19 utilisait le canal d'erreur standard pour afficher les chronométrages ; dans `Numerix`-0.21 toutes les sorties sont effectuées sur la sortie standard, y compris les messages d'erreurs internes.

2.4 Utilisation

2.4.1 Compilation

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

```
ocamlc   options nums.cma  numerix.cma  fichiers source
ocamlopt options nums.cmxa numerix.cmxa fichiers source
```

Les fichiers `nums.cma`, `nums.cmxa`, `numerix.cma` et `numerix.cmxa` contiennent sous forme compilée les bibliothèques `Big_int` et `Numerix`. Il peut être nécessaire d'indiquer aux compilateurs où trouver les fichiers `numerix.cma` et `numerix.cmxa` au moyen d'une option `-I` *chemin*.

2.4.2 Exemple

```
(* fichier simple.ml: démonstration de Numerix
   calcule (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
```

```
  let main arglist =

    let n = match arglist with
    | _::"-n"::x::_ -> int_of_string x
    | _             -> 30
    in

    (* d <- 10^n, d2 <- 10^(2n) *)
    let d  = (5 ^. n) << n in
    let d2 = sqr d         in

    (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2))) *)
    let a = gsqrt Nearest_up (d2 *. 200) in
    let b = gsqrt Nearest_up (d2 *. 300) in

    (* r <- round(10^n*(b+a)/(b-a)) *)
    let r = gquo Nearest_up (d**(b++a)) (b--a) in
    Printf.printf "r=%s\n" (string_of r);
    flush stdout

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

Compilation et exécution :

```
> ocamlc -I ~/lib -o simple-byte nums.cma numerix.cma simple.ml
> ./simple-byte -e slong
r=98989794855663561963945681494117827839318949613133 3
> ocamlopt -I ~/lib -o simple-opt nums.cmxa numerix.cmxa simple.ml
> ./simple-opt -e gmp -n 50 -count
r=98989794855663561963945681494117827839318949613133 3
   op      count   avg.size   max.size
  add          2        170        171
  mul          4        250        333
  quo          1        253        338
  pow          1          0          0
 root          2        340        341
  gcd          0          -          -
  bin          1        117        117
 misc          1        170        170
>
```

2.4.3 Système interactif

`ocamlnumx` est une version spécialisée du système interactif `ocaml` lié avec les fichiers objet `nums.cma` et `numerix.cma`. 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 : Ocaml toplevel with big integer libraries
Numerix submodules : Clong Dlong Slong Big Gmp
Numerix version    : 0.21

        Objective Caml version 3.08.0

# open Numerix;;
# module I = Infixes(Slong);; (* output deleted *)
# module R = Rfuns(Slong);;   (* output deleted *)
# open Slong open I open R;;
# let a = r_exp Floor one one (10^.50);;
val a : Numerix.Slong.t = 271828182845904523536028747135266249775724709369995
# #quit;;
>
```

Si votre version de Ocaml supporte les modules chargeables alors il est aussi possible d'utiliser le système interactif standard de Ocaml en chargeant explicitement les fichiers `nums.cma` et `numerix.cma`. Noter que dans ce cas, il peut être nécessaire d'indiquer à `ocaml` où trouver le fichier `numerix.cma` à l'aide d'une option `-I` *chemin*. Par ailleurs, les fonctions d'impression `toplevel_print` et `toplevel_print_tref` doivent être activées manuellement à l'aide de directives `#install_printer`.

```
> ocaml -I ~/lib
        Objective Caml version 3.08.0

# #load "nums.cma";;
# #load "numerix.cma";;
# open Numerix;;
# module I = Infixes(Slong);; (* output deleted *)
# module R = Rfuns(Slong);;   (* output deleted *)
# open Slong open I open R;;
# let a = r_exp Floor one one (10^.50);;
val a : Numerix.Slong.t = <abstr>
# #install_printer toplevel_print;;
# a;;
- : Numerix.Slong.t = 271828182845904523536028747135266249775724709369995
# #quit;;
>
```

Chapitre 3

Utilisation avec Camllight

Sommaire

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 0.74 et 0.75 de Camllight.

3.1 Interface

3.1.1 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 : `clong`, `dlong`, `slong`, `gmp` et `big`. Il n'y a pas d'équivalents aux modules construits en Ocaml avec les foncteurs `Cmp`, `Count` et `Rfuns`. Les notations infixes sont accessibles en ouvrant le module `infxxx` où `xxx` est le nom du module implémentant les grands entiers.

3.1.2 Fonctions

Les fonctions décrites dans la signature `Int_type` en Ocaml ont été conservées en Camllight avec seulement trois 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`, `gquo`, etc. 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.

- Les erreurs à l'exécution déclenchent une exception `Error` *msg* en Ocaml et `Failure "Numerix kernel :` *msg*`"` en Camllight. Ceci est dû à l'impossibilité de déclencher en Camllight depuis une fonction C une exception autre que `Failure` ou `Invalid_argument`.

3.2 Utilisation

3.2.1 Compilation

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

```
camlc -custom options nums.zo numerix.zo fichiers source \
      -lnumerix-caml -lnums -lgmp
```

Les fichiers `nums.zo` et `numerix.zo` contiennent sous forme compilée les parties Caml des bibliothèques `Big_int` et `Numerix`. Il peut être nécessaire d'indiquer au compilateur où trouver le fichier `numerix.zo` au moyen d'une option `-I` *chemin*.

Les options `-lnumerix-caml`, `-lnums` et `-lgmp` demandent à l'éditeur de liens de rechercher dans les bibliothèques `libnumerix-caml`, `libnums` et `libgmp` les primitives C dont il aurait besoin. Il peut être nécessaire d'indiquer à l'éditeur de liens dans quels répertoires chercher ces fichiers 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 `-lnums` doivent être omis si le module `big` n'est pas inclus dans `Numerix`. Noter que la bibliothèque `libclnumx` utilisée avec `Numerix`-0.19 a été renommée `libnumerix-caml` dans `Numerix`-0.21, le nouveau nom étant jugé plus parlant.

3.2.2 Exemple

```
(* fichier simple.ml: démonstration de Numerix
   calcule (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales *)
```

```
#open "clong";;
#open "infclong";;

  let main arglist =

    let n = match arglist with
    | _::"-n"::x::_ -> int_of_string x
    | _             -> 30
    in

    (* d <- 10^n, d2 <- 10^(2n) *)
    let d  = (5 ^. n) << n in
    let d2 = sqr d         in

    (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2))) *)
    let a = gsqrt Nearest_up (d2 *. 200) in
    let b = gsqrt Nearest_up (d2 *. 300) in

    (* r <- round(10^n*(b+a)/(b-a)) *)
    let r = gquo Nearest_up (d**(b++a)) (b--a) in
    printf__printf "r=%s\n" (string_of r);
    flush stdout

  in
  main (list_of_vect sys__command_line);;
```

Compilation et exécution :

```
> camlc -custom -I ~/lib -o simple nums.zo numerix.zo simple.ml \
        -lnumerix-caml -lnums -lgmp -ccopt -L/home/quercia/lib
> ./simple
r=98989794855663561963945681494l1
>
```

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

3.2.3 Système interactif

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

```
> camllight ~/lib/camlnumx
>       Caml Light version 0.75

camlnumx : Caml toplevel with big integer libraries
```

```
Numerix submodules : clong dlong slong big gmp
Numerix version    : 0.21

##open "slong";;
##open "infslong";;
#fact 30;;
- : t = 265252859812191058636308480000000
#one << 100;;
- : t = 1267650600228229401496703205376
#quit();;
>
```

Chapitre 4

Utilisation en C

Sommaire

4.1 Interface

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 implémentées par le noyau C de `Numerix`. 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. L'interface C de `Numerix` a été testée avec les compilateurs `gcc-3.3.3`, `gcc-2.95.3`, `gcc-2.7.2.3` et les systèmes d'exploitation Linux, OpenBSD et Digital Unix.

4.1.1 Conventions

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 doit être indiqué à la compilation au moyen d'une directive `#define use_xxx` où `xxx` est le nom du module désiré. Cette directive peut être incluse en tête de chaque fichier source ou transmise au préprocesseur à l'aide d'une option `-Duse_xxx` sur la ligne de commande de compilation. Noter les différences suivantes par rapport à `Numerix`-0.19 :

– il n'est plus nécessaire d'indiquer la taille d'un mot machine ;
– il n'y a pas de module sélectionné par défaut ;
– le fichier d'en-tête décrivant `Numerix`-0.19 était nommé `c-long_int.h`, celui décrivant `Numerix`-0.21 est nommé `numerix.h`.

Le fichier `numerix.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 auquel elles appartiennent : `cx_` pour le module `clong`, `dx_` pour `dlong` et `sx_` pour `slong`. De façon à permettre l'écriture d'un code indépendant de l'implémentation des grands entiers le fichier `numerix.h` définit une macro `xx` qui accole à son argument le préfixe `cx_`, `dx_` ou `sx_` en fonction du symbole `use_clong`, `use_dlong` ou `use_slong` qui 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.

En règle générale, une fonction calculant un résultat `a` de type `xint` est fournie en deux versions différant par leur convention d'appel :

`xint xx(func)(xint *_a,` *args*`)`
`xint xx(f_func)(`*args*`)`

Dans les deux cas, la valeur de retour est le résultat `a` calculé. De plus, si `_a != NULL`, alors le résultat est copié à l'emplacement désigné par `_a`. Il y a deux exceptions à cette convention de nommage : `copy_int` et `copy_string` ont pour fonctions associées les fonctions `of_int` et `of_string` au lieu de `f_copy_int` et `f_copy_string` par compatibilité avec `Numerix`-0.19. Une fonction calculant plusieurs résultats `a`, `b`,... de type `xint` est fournie en une seule version :

`void xx(func)(xint *_a, xint *_b,...,` *args)*

Les résultats `a`, `b`,... calculés sont copiés aux emplacements désignés par les pointeurs `_a`, `_b`,... Si l'un des pointeurs est `NULL`, le résultat correspondant n'est pas copié et n'est donc pas accessible à l'appelant.

4.1.2 Le fichier `numerix.h`

Voici un extrait de `numerix.h` donnant les prototypes des fonctions publiques :

```
typedef struct {...} *xint;

/*-------------------- création/destruction */
xint xx(new)();
void xx(free)(xint *_x);

xint xx(copy)   (xint *_b, xint a);
xint xx(f_copy) (xint a);

/*-------------------- addition/soustraction */
xint xx(add)    (xint *_c, xint a, xint b);
xint xx(sub)    (xint *_c, xint a, xint b);
xint xx(add_1)  (xint *_c, xint a, long b);
xint xx(sub_1)  (xint *_c, xint a, long b);

xint xx(f_add)  (xint a, xint b);
xint xx(f_sub)  (xint a, xint b);
xint xx(f_add_1)(xint a, long b);
xint xx(f_sub_1)(xint a, long b);

/*-------------------- multiplication/carré */
xint xx(mul)    (xint *_c, xint a, xint b);
xint xx(mul_1)  (xint *_c, xint a, long b);
xint xx(sqr)    (xint *_b, xint a);

xint xx(f_mul)  (xint a, xint b);
xint xx(f_mul_1)(xint a, long b);
xint xx(f_sqr)  (xint a);

/*-------------------- division */
void xx(quomod)   (xint *_c, xint *_d, xint a, xint b);
xint xx(quo)      (xint *_c,           xint a, xint b);
xint xx(mod)      (xint *_d,           xint a, xint b);
long xx(quomod_1) (xint *_c,           xint a, long b);
xint xx(quo_1)    (xint *_c,           xint a, long b);
long xx(mod_1)    (                    xint a, long b);
void xx(gquomod)  (xint *_c, xint *_d, xint a, xint b, long mode);
xint xx(gquo)     (xint *_c,           xint a, xint b, long mode);
xint xx(gmod)     (xint *_d,           xint a, xint b, long mode);
long xx(gquomod_1)(xint *_c,           xint a, long b, long mode);
xint xx(gquo_1)   (xint *_c,           xint a, long b, long mode);
long xx(gmod_1)   (                    xint a, long b, long mode);

xint xx(f_quo)    (xint a, xint b);
xint xx(f_mod)    (xint a, xint b);
xint xx(f_quo_1)  (xint a, long b);
long xx(f_mod_1)  (xint a, long b);
xint xx(f_gquo)   (xint a, xint b, long mode);
xint xx(f_gmod)   (xint a, xint b, long mode);
xint xx(f_gquo_1) (xint a, long b, long mode);
```

```
long xx(f_gmod_1) (xint a, long b, long mode);

/*-------------------- valeur absolue, opposé */
xint xx(abs)    (xint *_b, xint a);
xint xx(neg)    (xint *_b, xint a);
xint xx(f_abs)  (xint a);
xint xx(f_neg)  (xint a);

/*-------------------- exponentiation */
xint xx(pow)      (xint *_b, xint a, long p);
xint xx(pow_1)    (xint *_b, long a, long p);
xint xx(powmod)   (xint *_d, xint a, xint b, xint c);
xint xx(gpowmod)  (xint *_d, xint a, xint b, xint c, long mode);

xint xx(f_pow)    (xint a, long p);
xint xx(f_pow_1)  (long a, long p);
xint xx(f_powmod) (xint a, xint b, xint c);
xint xx(f_gpowmod)(xint a, xint b, xint c, long mode);

/*-------------------- racines */
xint xx(sqrt)   (xint *_b, xint a);
xint xx(root)   (xint *_b, xint a, long p);
xint xx(gsqrt)  (xint *_b, xint a,         long mode);
xint xx(groot)  (xint *_b, xint a, long p, long mode);

xint xx(f_sqrt) (xint a);
xint xx(f_root) (xint a, long p);
xint xx(f_gsqrt)(xint a,         long mode);
xint xx(f_groot)(xint a, long p, long mode);

/*-------------------- factorielle */
xint xx(fact)  (xint *_a, long n);
xint xx(f_fact)(long n);

/*-------------------- pgcd */
xint xx(gcd)   (xint *_d, xint a, xint b);
void xx(gcd_ex)(xint *_d, xint *_u, xint *_v, xint a, xint b);
void xx(cfrac) (xint *_d, xint *_u, xint *_v, xint *_p, xint *_q, xint a, xint b);
xint xx(f_gcd) (xint a, xint b);

/*-------------------- comparaison */
long xx(sgn)    (xint a);
long xx(cmp)    (xint a, xint b);
long xx(cmp_1)  (xint a, long b);

long xx(eq)     (xint a,xint b);
long xx(neq)    (xint a,xint b);
long xx(inf)    (xint a,xint b);
long xx(infeq)  (xint a,xint b);
long xx(sup)    (xint a,xint b);
```

```
long xx(supeq)  (xint a,xint b);

long xx(eq_1)   (xint a,long b);
long xx(neq_1)  (xint a,long b);
long xx(inf_1)  (xint a,long b);
long xx(infeq_1)(xint a,long b);
long xx(sup_1)  (xint a,long b);
long xx(supeq_1)(xint a,long b);

/*-------------------- conversion */
xint xx(copy_int)   (xint *_b, long a);
xint xx(of_int)     (long a);
long xx(int_of)     (xint a);
xint xx(copy_string)(xint *_a, char *s);
xint xx(of_string)  (char *s);

char *xx(string_of) (xint a);
char *xx(hstring_of)(xint a);
char *xx(ostring_of)(xint a);
char *xx(bstring_of)(xint a);

/*-------------------- nombres aléatoires */
void xx(random_init)(long n);
xint xx(nrandom) (xint *_a, long n);
xint xx(zrandom) (xint *_a, long n);
xint xx(nrandom1)(xint *_a, long n);
xint xx(zrandom1)(xint *_a, long n);

xint xx(f_nrandom) (long n);
xint xx(f_zrandom) (long n);
xint xx(f_nrandom1)(long n);
xint xx(f_zrandom1)(long n);

/*-------------------- représentation binaire */
long xx(nbits)   (xint a);
long xx(lowbits) (xint a);
long xx(highbits)(xint a);
long xx(nth_word)(xint a, long n);
long xx(nth_bit) (xint a, long n);

/*-------------------- décalages */
xint xx(shl)  (xint *_b,           xint a, long n);
xint xx(shr)  (xint *_b,           xint a, long n);
void xx(split)(xint *_b, xint *_c, xint a, long n);
xint xx(join) (xint *_c, xint a,   xint b, long n);

xint xx(f_shl) (xint a,         long n);
xint xx(f_shr) (xint a,         long n);
xint xx(f_join)(xint a, xint b, long n);
```

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

4.1.3 Gestion de la mémoire

Une variable `a` 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 de `a` s'effectue normalement en deux étapes :

- initialisation du pointeur `a` ;
- attribution d'une valeur en indiquant l'adresse `&a` en paramètre résultat d'un calcul.

Il est possible de combiner ces deux étapes en une seule en affectant à `a` le résultat de type `xint` retourné par un calcul. Ainsi, les séquences suivantes où `a` désigne une variable de type `xint` non initialisée et `b`,`c` désignent des variables de type `xint` initialisées ayant reçu des valeurs b et c sont équivalentes : leur effet commun est d'allouer un bloc mémoire, d'y copier la représentation interne du nombre $b + c$, et de copier l'adresse de ce bloc dans `a`.

```
a = xx(new)(); xx(add)(&a,b,c);
a = xx(f_add)(b,c);
a = xx(add)(NULL,b,c);
```

Une fois que le pointeur `a` est initialisé, l'adresse `&a` peut être passée en paramètre résultat d'un calcul. Par exemple :

```
xx(mul)(&a,b,c);
```

a pour effet de calculer le produit bc et de copier dans `a` l'adresse du bloc mémoire où ce produit a été stocké. Il n'est pas nécessaire que `a` ait reçu une valeur préalablement à cette opération. Si c'est le cas alors le bloc mémoire contenant cette valeur est surchargé avec la représentation interne de bc s'il est suffisamment grand, sinon un nouveau bloc mémoire est alloué pour recevoir le résultat, `a` est modifié pour pointer vers ce nouveau bloc et l'ancien bloc est libéré. 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`, `gquomod`, `gcd_ex`, `cfrac` et `split`) une même variable ne peut figurer plusieurs fois en position résultat. Ainsi l'instruction suivante est illégale :

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

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

```
xx(free)(&a);
```

a pour effet de libérer le bloc mémoire pointé par `a` s'il y en a un et de réinitialiser le pointeur `a`. La variable `a` reste utilisable après cette instruction pour recevoir une nouvelle valeur.

4.1.4 Mode d'arrondi

Les opérations produisant une approximation entière d'un nombre réel a (division, racine carrée et racine p-ème) sont fournies en deux versions :

`xx(func)` (*args*)
`xx(gfunc)`(*args*, `long mode`)

Le paramètre `mode` de `xx(gfunc)` indique de quelle manière arrondir le nombre a :

si `mode` & $3 = 0$: calculer $\lfloor a \rfloor$;
si `mode` & $3 = 1$: calculer $\lfloor a + 1/2 \rfloor$;
si `mode` & $3 = 2$: calculer $\lceil a \rceil$;
si `mode` & $3 = 3$: calculer $\lceil a - 1/2 \rceil$.

`xx(func)` est équivalent à `xx(gfunc)` avec `mode` $= 0$.

4.1.5 Description des fonctions

Les opérations implémentées dans l'interface C de `Numerix` sont identiques à celles implémentées dans l'interface Ocaml et décrites dans les sections **2.2.4 Opérations arithmétiques** à **2.2.8 Accès à la représentation binaire**, pages 14 et suivantes et dans la section **2.3.6 Chronométrage**, page 24. Seules sont données ici les particularités propres à l'interface C.

- Lorsqu'une fonction Ocaml retourne un résultat booléen, son équivalent C retourne un entier de type `long` valant 0 pour `false` et 1 pour `true`.

- Les fonctions C convertissant un grand entier en chaîne de caractères retournent un pointeur vers une chaîne allouée sur le tas. Cette chaîne devra être libérée après utilisation par appel à la fonction `free`. Noter que `Numerix`-0.19 fournissait une fonction `xx(free_string)` à cet effet ; cette fonction n'existe plus dans `Numerix`-0.21.

- Les fonctions `xx(lowbits)` et `xx(highbits)` retournent les 31 bits de poids faible ou fort de leur argument, indépendamment de la taille d'un mot machine. De même, la fonction `xx(int_of)` déclenche systématiquement une erreur si la valeur absolue de son argument est supérieure ou égale à 2^{30}.

4.2 Utilisation

4.2.1 Compilation

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

`gcc` *options* `-Duse_xxx` *fichiers source* `-lnumerix-c`

`-Duse_xxx` indique le module à utiliser, `clong` ou `dlong` ou `slong`.

`-lnumerix-c` indique à l'éditeur de liens de rechercher dans la bibliothèque `libnumerix-c` les fonctions compilées dont il aurait besoin. Il peut être nécessaire d'indiquer à l'éditeur de liens où trouver ce fichier 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 `numerix.h`. Noter que la bibliothèque `libcnumx` utilisée avec `Numerix`-0.19 a été renommée `libnumerix-c` dans `Numerix`-0.21, le nouveau nom étant jugé plus parlant.

4.2.2 Exemple

```
/* fichier simple.c: démonstration de Numerix
   calcule (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "numerix.h"

int main(int argc, char **argv) {

   xint a,b,d,d2,x,y;
   char *s;
   long n;

   /* nombre de décimales */
   if ((argc > 2) && (strcmp(argv[1],"-n") == 0)) n = atol(argv[2]);
   else n = 30;

   /* d <- 10^n, d2 <- 10^(2n) */
   d  = xx(f_pow_1)(5,n); xx(shl)(&d,d,n);
   d2 = xx(f_sqr)(d);

   /* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2)))  */
   a = xx(f_mul_1)(d2,200); xx(gsqrt)(&a,a,1);
   b = xx(f_mul_1)(d2,300); xx(gsqrt)(&b,b,1);

   /* x <- round(10^n*(b+a)/(b-a)) */
   x = xx(f_add)(b,a); xx(mul)(&x,x,d);
   y = xx(f_sub)(b,a);
   xx(gquo)(&x,x,y,1);

   /* affiche x */
   s = xx(string_of)(x); printf("x=%s\n",s); free(s);

   /* libère la mémoire temporaire */
   xx(free)(&d); xx(free)(&d2);
   xx(free)(&a); xx(free)(&b);
   xx(free)(&x); xx(free)(&y);

   return(0);
```

```
}
```

Compilation et exécution :

```
> gcc -O2 -Wall -I/home/quercia/include -Duse_slong \
      -o simple simple.c -lnumerix-c -L/home/quercia/lib
> ./simple -n 20
x=989897948556635619642
>
```

Chapitre 5

Utilisation en Pascal

Sommaire

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 avec le compilateur Free Pascal version 1.0.10.

5.1 Interface

5.1.1 Unités

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.

Voici un extrait du fichier `clong.p` décrivant l'unité `clong` :

```
unit clong;
interface

type xint = ...;

(* création/destruction *)
function xnew : xint;
```

```
procedure xfree(var x : xint);

procedure copy (var b:xint; a:xint);
function f_copy(a:xint):xint;

(* addition/soustraction *)
procedure add  (var c:xint; a:xint; b:xint   );
procedure sub  (var c:xint; a:xint; b:xint   );
procedure add_1(var c:xint; a:xint; b:longint);
procedure sub_1(var c:xint; a:xint; b:longint);

function f_add  (a:xint; b:xint   ):xint;
function f_sub  (a:xint; b:xint   ):xint;
function f_add_1(a:xint; b:longint):xint;
function f_sub_1(a:xint; b:longint):xint;

(* multiplication *)
procedure mul  (var c:xint; a:xint; b:xint   );
procedure mul_1(var c:xint; a:xint; b:longint);
procedure sqr  (var b:xint; a:xint);

function f_mul  (a:xint; b:xint   ):xint;
function f_mul_1(a:xint; b:longint):xint;
function f_sqr  (a:xint):xint;

(* division *)
procedure quomod   (var c,d:xint; a:xint; b:xint   );
procedure quo      (var c  :xint; a:xint; b:xint   );
procedure modulo   (var d  :xint; a:xint; b:xint   );
procedure quomod_1 (var c  :xint; a:xint; b:longint);
procedure quo_1    (var c  :xint; a:xint; b:longint);
function  mod_1                  (a:xint; b:longint):longint;
procedure gquomod  (var c,d:xint; a:xint; b:xint;    mode:longint);
procedure gquo     (var c  :xint; a:xint; b:xint;    mode:longint);
procedure gmod     (var d  :xint; a:xint; b:xint;    mode:longint);
function  gquomod_1(var c  :xint; a:xint; b:longint; mode:longint):longint;
procedure gquo_1   (var c  :xint; a:xint; b:longint; mode:longint);
function  gmod_1                 (a:xint; b:longint; mode:longint):longint;

function f_quo     (a:xint; b:xint   ):xint;
function f_mod     (a:xint; b:xint   ):xint;
function f_quo_1   (a:xint; b:longint):xint;
function f_mod_1   (a:xint; b:longint):longint;
function f_gquo    (a:xint; b:xint;    mode:longint):xint;
function f_gmod    (a:xint; b:xint;    mode:longint):xint;
function f_gquo_1  (a:xint; b:longint; mode:longint):xint;
function f_gmod_1  (a:xint; b:longint; mode:longint):longint;

(* valeur absolue/opposé *)
procedure abs  (var b:xint; a:xint);
```

```
procedure neg  (var b:xint; a:xint);

function f_abs (a:xint):xint;
function f_neg (a:xint):xint;

(* exponentiation *)
procedure pow    (var b:xint; a:xint;    p:longint);
procedure pow_1  (var b:xint; a:longint; p:longint);
procedure powmod (var d:xint; a:xint;    b:xint; c:xint);
procedure gpowmod(var d:xint; a:xint;    b:xint; c:xint; mode:longint);

function f_pow    (a:xint;    p:longint):xint;
function f_pow_1  (a:longint; p:longint):xint;
function f_powmod (a:xint;    b:xint; c:xint):xint;
function f_gpowmod(a:xint;    b:xint; c:xint; mode:longint):xint;

(* racines *)
procedure sqrt (var b:xint; a:xint);
procedure gsqrt(var b:xint; a:xint;            mode:longint);
procedure root (var b:xint; a:xint; p:longint);
procedure groot(var b:xint; a:xint; p:longint; mode:longint);

function f_sqrt (a:xint                ):xint;
function f_gsqrt(a:xint; mode:  longint):xint;
function f_root (a:xint; p:     longint):xint;
function f_groot(a:xint; p,mode:longint):xint;

(* factorielle *)
procedure fact(var a:xint; n:longint);
function f_fact(n:longint):xint;

(* pgcd *)
procedure gcd   (var d:xint;         a,b:xint);
procedure gcd_ex(var d,u,v:xint;     a,b:xint);
procedure cfrac (var d,u,v,p,q:xint; a,b:xint);

function f_gcd(a,b:xint):xint;

(* comparaison *)
function cmp    (a:xint; b:xint   ):longint;
function cmp_1  (a:xint; b:longint):longint;
function sgn    (a:xint           ):longint;
function eq     (a:xint; b:xint   ):boolean;
function neq    (a:xint; b:xint   ):boolean;
function inf    (a:xint; b:xint   ):boolean;
function infeq  (a:xint; b:xint   ):boolean;
function sup    (a:xint; b:xint   ):boolean;
function supeq  (a:xint; b:xint   ):boolean;
function eq_1   (a:xint; b:longint):boolean;
function neq_1  (a:xint; b:longint):boolean;
```

```
function inf_1  (a:xint; b:longint):boolean;
function infeq_1(a:xint; b:longint):boolean;
function sup_1  (a:xint; b:longint):boolean;
function supeq_1(a:xint; b:longint):boolean;

(* conversions *)
procedure copy_int(var b:xint; a:longint);
procedure copy_string(var a:xint; s:pchar);

function of_int(a:longint):xint;
function of_string(s:pchar):xint;

function string_of (a:xint):ansistring;
function hstring_of(a:xint):ansistring;
function ostring_of(a:xint):ansistring;
function bstring_of(a:xint):ansistring;

(* nombres aléatoires *)
procedure random_init(n:longint);

procedure nrandom (var a:xint; n:longint);
procedure zrandom (var a:xint; n:longint);
procedure nrandom1(var a:xint; n:longint);
procedure zrandom1(var a:xint; n:longint);

function f_nrandom (n:longint):xint;
function f_zrandom (n:longint):xint;
function f_nrandom1(n:longint):xint;
function f_zrandom1(n:longint):xint;

(* représentation binaire *)
function int_of  (a:xint           ):longint;
function nbits   (a:xint           ):longint;
function lowbits (a:xint           ):longint;
function highbits(a:xint           ):longint;;
function nth_word(a:xint; n:longint):longint;;
function nth_bit (a:xint; n:longint):boolean;

(* décalages *)
procedure shiftl(var b:xint; a:xint; n:longint);
procedure shiftr(var b:xint; a:xint; n:longint);
procedure split(var b,c:xint; a:xint; n:longint);
procedure join(var c:xint; a:xint; b:xint; n:longint);

function f_shl(a:xint; n:longint):xint;
function f_shr(a:xint; n:longint):xint;
function f_join(a:xint; b:xint; n:longint):xint;

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

5.1.2 Gestion de la mémoire

Une variable `a` 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 de `a` s'effectue normalement en deux étapes :

- initialisation du pointeur `a` ;
- attribution d'une valeur en passant `a` en paramètre résultat d'un calcul.

Il est possible de combiner ces deux étapes en une seule en affectant à `a` le résultat de type `xint` retourné par un calcul. Ainsi, les séquences suivantes où `a` désigne une variable de type `xint` non initialisée et `b`,`c` désignent des variables de type `xint` initialisées ayant reçu des valeurs b et c sont équivalentes : leur effet commun est d'allouer un bloc mémoire, d'y copier la représentation interne du nombre $b + c$, et de copier l'adresse de ce bloc dans `a`.

```
a := xnew(); add(a,b,c);
a := f_add(b,c);
```

Une fois que le pointeur `a` est initialisé, `a` peut être passé en paramètre résultat d'un calcul. Par exemple :

```
mul(a,b,c);
```

a pour effet de calculer le produit bc et de copier dans `a` l'adresse du bloc mémoire où ce produit a été stocké. Il n'est pas nécessaire que `a` ait reçu une valeur préalablement à cette opération. Si c'est le cas alors le bloc mémoire contenant cette valeur est surchargé avec la représentation interne de bc s'il est suffisamment grand, sinon un nouveau bloc mémoire est alloué pour recevoir le résultat, `a` est modifié pour pointer vers ce nouveau bloc et l'ancien bloc est libéré. 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`, `gquomod`, `gcd_ex`, `cfrac` et `split`) une même variable ne peut figurer plusieurs fois en position résultat. Ainsi l'instruction suivante est illégale :

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

La procédure `xfree` permet de libérer un bloc mémoire occupé par une valeur qui n'est plus utile. L'instruction :

```
xfree(a);
```

a pour effet de libérer le bloc mémoire pointé par `a` s'il y en a un et de réinitialiser le pointeur `a`. La variable `a` reste utilisable après cette instruction pour recevoir une nouvelle valeur.

5.1.3 Mode d'arrondi

Les opérations produisant une approximation entière d'un nombre réel a (division, racine carrée et racine p-ème) sont fournies en deux versions :

`func (`*args*`)`
`gfunc(`*args*`; mode:longint)`

Le paramètre `mode` de `gfunc` indique de quelle manière arrondir le nombre a :

si `mode and` $3 = 0$: calculer $\lfloor a \rfloor$;
si `mode and` $3 = 1$: calculer $\lfloor a + 1/2 \rfloor$;
si `mode and` $3 = 2$: calculer $\lceil a \rceil$;
si `mode and` $3 = 3$: calculer $\lceil a - 1/2 \rceil$.

`func` est équivalent à `gfunc` avec `mode` $= 0$.

5.1.4 Description des fonctions

Les opérations implémentées dans l'interface Pascal de `Numerix` sont identiques à celles implémentées dans l'interface Ocaml et décrites dans les sections **2.2.4 Opérations arithmétiques** à **2.2.8 Accès à la représentation binaire**, pages 14 et suivantes et dans la section **2.3.6 Chronométrage**, page 24. Seules sont données ici les particularités propres à l'interface Pascal.

– Les fonctions `lowbits` et `highbits` retournent les 31 bits de poids faible ou fort de leur argument, indépendamment de la taille d'un mot machine. De même, la fonction `int_of` déclenche systématiquement une erreur si la valeur absolue de son argument est supérieure ou égale à 2^{30}.

5.2 Utilisation

5.2.1 Compilation

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

`fpc` *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 `libnumerix-c`. Ces directives peuvent être omises si ces fichiers sont stockés dans les répertoires normalement examinés par le compilateur Pascal. Noter que la bibliothèque `libcnumx` utilisée avec `Numerix`-0.19 a été renommée `libnumerix-c` dans `Numerix`-0.21, le nouveau nom étant jugé plus parlant.

5.2.2 Exemple

```
program simple;
(* fichier simple.p: démonstration de Numerix
```

```
   calcule (sqrt(3) + sqrt(2))/(sqrt(3)-sqrt(2)) avec n décimales *)

uses clong;

var a,b,d,d2,x,y:xint;
    n : longint;
    c : word;
begin

   (* nombre de décimales *)
   if (paramcount >= 2) and (paramstr(1) = '-n')
      then val(paramstr(2),n,c)
      else n := 30;

   (* d <- 10^n, d2 <- 10^(2n) *)
   d  := f_pow_1(5,n); shiftl(d,d,n);
   d2 := f_sqr(d);

   (* a <- round(sqrt(2*10^(2n+2))), b <- round(sqrt(3*10^(2n+2)))  *)
   a := f_mul_1(d2,200); gsqrt(a,a,1);
   b := f_mul_1(d2,300); gsqrt(b,b,1);

   (* x <- round(10^n*(b+a)/(b-a)) *)
   x := f_add(b,a); mul(x,x,d);
   y := f_sub(b,a);
   gquo(x,x,y,1);

   (* affiche x *)
   writeln('x=',string_of(x));

   (* libère la mémoire temporaire *)
   xfree(d); xfree(d2);
   xfree(a); xfree(b);
   xfree(x); xfree(y);

end.
```

Compilation et exécution :

```
> fpc -Fu/home/quercia/lib -Fl/home/quercia/lib simple.p
Free Pascal Compiler version 1.0.10 [2003/06/26] for i386
Copyright (c) 1993-2003 by Florian Klaempfl
Target OS: Linux for i386
Compiling simple.p
Assembling simple
Linking simple
38 Lines compiled, 0.0 sec
> ./simple -n 50
x=98989794855663561963945681494117827839318949613133 3
>
```

Chapitre 6

Installation

Sommaire

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 `gcc-3.3.3` 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 `3.06` et pour Camllight d'une version supérieure ou égale à `0.74`. 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/`

6.2 Configuration

6.2.1 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` lancer :

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

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

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

Ce script détermine quelles parties de `Numerix` peuvent être compilées sur votre machine et crée un fichier `Makefile` 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`.
dir doit désigner un chemin absolu. 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. *dir* doit désigner un chemin absolu.

`--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 Free Pascal 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`.

`--enable-x86`, `--disable-x86`

Indique si le processeur est ou n'est pas compatible `x86`. Par défaut le processeur est supposé être de type `x86` si le nom canonique du système d'exploitation contient le motif `i*86`.

`--enable-sse2`, `--disable-sse2`

Indique si le processeur, qui doit être de type `x86`, dispose ou non du jeu d'instructions `SSE2`. Par défaut, si le processeur est de type `x86` et le système d'exploitation contient la chaîne `Linux`, le programme `configure` consulte le fichier `/proc/cpuinfo` pour obtenir ce renseignement.

`--enable-alloca`, `--disable-alloca`

Indique si la fonction d'allocation temporaire `alloca` peut être utilisée à la place de `malloc`. Par défaut l'usage de `alloca` est activé si `configure` a trouvé une interface correcte pour cette fonction.

6.2.2 Configuration manuelle

Normalement le script `configure` décrit à la section précédente devrait créer des fichiers `Makefile`, `kernel/*/makefile` et `kernel/config.h` convenables.

En cas de problème éditez les fichiers `Makefile` et `kernel/config.h` pour corriger les valeurs indiquées par `configure` si elles sont incorrectes. Après correction, vous devez recréer les fichiers auxiliaires `kernel/*/makefile` pour prendre en compte les modifications effectuées et effacer les fichiers créés par une compilation précédente. Pour ce faire lancez les commandes :

```
make makefiles
make clean
```

6.2.3 Édition du fichier `Makefile`

Utilisez les valeurs 0 ou 1 pour les paramètres devant recevoir une valeur booléenne (1 = vrai).

```
PROCESSOR = x86-sse2
```

Indiquez le type de processeur : `x86-sse2` pour un processeur de type `x86` disposant du jeu d'instructions `SSE2`, `x86` pour un processeur de type `x86` n'en disposant pas, et `generic` pour les autres processeurs.

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

Indiquez quelles interfaces vous voulez compiler.

```
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 `Big_int`.

```
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.

```
CAML_LIBDIR = /usr/local/lib/caml-light
CAMLC       = camlc
CAMLLIBR    = camllibr
CAMLMKTOP   = camlmktop
```

Indiquez le répertoire dans lequel Camllight est installé et quelles sont les commandes à lancer pour invoquer respectivement le compilateur Camllight, l'archiveur Camllight et le compilateur de toplevel Camllight.

```
OCAML_LIBDIR = /usr/local/lib/ocaml
OCAMLC       = ocamlc
OCAMLOPT     = ocamlopt
OCAMLMKTOP   = ocamlmktop
OCAMLMKLIB   = ocamlmklib
```

Indiquez le répertoire dans lequel Ocaml est installé et quelles sont les commandes à lancer pour invoquer respectivement le compilateur Ocaml, le compilateur optimiseur Ocaml, le compilateur de système interactif Ocaml et le compilateur de bibliothèques Ocaml.

```
FPC = fpc -v0 -k-lgcc_s
```

Indiquez la commande à lancer pour invoquer le compilateur Pascal.

```
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 et les programmes exécutables.

```
C_INSTALL_BIN          = $(INSTALL_BIN)
C_INSTALL_LIB          = $(INSTALL_LIB)
C_INSTALL_INCLUDE      = $(INSTALL_INCLUDE)

CAML_INSTALL_BIN       = $(INSTALL_BIN)
CAML_INSTALL_LIB       = $(INSTALL_LIB)
CAML_INSTALL_INCLUDE   = $(INSTALL_INCLUDE)

OCAML_INSTALL_BIN      = $(INSTALL_BIN)
OCAML_INSTALL_LIB      = $(INSTALL_LIB)
OCAML_INSTALL_INCLUDE  = $(INSTALL_INCLUDE)

PASCAL_INSTALL_BIN     = $(INSTALL_BIN)
PASCAL_INSTALL_LIB     = $(INSTALL_LIB)
PASCAL_INSTALL_INCLUDE = $(INSTALL_INCLUDE)
```

Normalement les mêmes répertoires désignés par `INSTALL_BIN`, `INSTALL_LIB` et `INSTALL_INCLUDE` sont utilisés pour tous les langages sélectionnés. Vous pouvez définir un jeu de répertoires différents pour chaque langage en modifiant les paramètres correspondants. Noter que les valeurs de `CAML_INSTALL_LIB` et `OCAML_INSTALL_LIB` sont copiées en dur dans les systèmes interactifs `ocamlnumx` et `camlnumx` de façon que ces systèmes puissent trouver par eux-mêmes les interfaces compilées `numerix.cmi` et `numerix.zi`, donc si vous voulez ultérieurement déplacer ces répertoires, vous devrez recompiler `camlnumx` et `ocamlnumx`.

6.2.4 Édition du fichier `kernel/config.h`

Ce fichier contient les réglages internes au noyau C/assembleur de `Numerix`. Il est normalement créé par le script `configure` en fonction du type de processeur détecté et de la possibilité d'utiliser la fonction `alloca`. Si `configure` détecte mal ces informations, utilisez les options `--enable_xxx` et `--disable_xxx` décrites à la section 6.2.1 pour imposer des valeurs correctes. Si `configure` ne fonctionne pas, copiez l'un des fichiers `config/generic.h`, `config/x86.h` ou `config/x86-sse2.h` sur `kernel/config.h`, puis éditez ce dernier pour indiquer la taille en bits d'un mot machine et s'il faut ou non utiliser `alloca` :

```
/* Machine word size */
#define bits_@machine_word_size@

/* Memory allocation strategy  */
@use_alloca@
```

Remplacez la chaîne `@machine_word_size@` par 32 ou 64 et `@use_alloca@` par `#define use_alloca` ou `#undef use_alloca`.

6.3 Compilation

Après configuration automatique ou manuelle 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 `conflog`, `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` inclus dans la distribution contiennent les traces de compilation sur un PC-Linux, 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
```

Voir page 54 la liste des fichiers installés, classée par répertoire. Seuls sont installés les fichiers correspondant aux modules et langages sélectionnés et au choix statique/dynamique des bibliothèques.

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/francais` au format PDF (`numerix.pdf`) et au format LaTeX (`numerix.tex`) si vous désirez l'imprimer.

6.4 Description des exemples

Les sous-répertoires `c`,`caml`,`ocaml` et `pascal` du répertoire `exemples` contiennent divers programmes utilisant `Numerix`. Pour compiler ces programmes lancer la commande :

```
make exemples
```

Dans le cas des exemples en C, Caml et Pascal, un même fichier source `ex.`*ext* est compilé en autant d'exécutables qu'il y a de modules disponibles implémentant les grands entiers pour ce langage. Chaque exécutable est nommé `ex-x` où `x` est l'initiale du module de grands entiers utilisé. Dans le cas des exemples en Ocaml, un même fichier source `ex.ml` est compilé en deux exécutables : `ex` avec le compilateur `ocamlc` et `ex-opt` avec le compilateur `ocamlopt`. Le choix d'un module de grands entiers s'effectue à l'exécution au moyen d'une option `-e xxx` comme décrit à la section **2.3.5 Sélection d'un module à l'exécution**, page 23.

6.4.1 chrono

Mesure de vitesse des différentes bibliothèques (interface 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 ;

FIG. 6.1 – liste des fichiers `Numerix` installés

$(C_INSTALL_LIB)	$(CAML_INSTALL_LIB)	$(OCAML_INSTALL_LIB)	$(PASCAL_INSTALL_LIB)
libnumerix-c.a	libnumerix-caml.a	libnumerix-ocaml.a dllnumerix-ocaml.so	
	numerix.zo camlnumx big.zi clong.zi dlong.zi gmp.zi slong.zi infbig.zi infclong.zi infdlong.zi infgmp.zi infslong.zi	numerix.a numerix.cma numerix.cmi numerix.cmxa	clong.o clong.ppu dlong.o dlong.ppu slong.o slong.ppu

$(C_INSTALL_INCLUDE)	$(CAML_INSTALL_INCLUDE)	$(OCAML_INSTALL_INCLUDE)	$(PASCAL_INSTALL_INCLUDE)
numerix.h	big.ml big.mli clong.ml clong.mli dlong.ml dlong.mli gmp.ml gmp.mli slong.ml slong.mli infbig.ml infbig.mli infclong.ml infclong.mli infdlong.ml infdlong.mli infgmp.ml infgmp.mli infslong.ml infslong.mli	numerix.ml numerix.mli	clong.p dlong.p slong.p

$(C_INSTALL_BIN)	$(CAML_INSTALL_BIN)	$(OCAML_INSTALL_BIN)	$(PASCAL_INSTALL_BIN)
		ocamlnumx	

`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.

```
> exemples/c/chrono-s -all 1000000 -r 10
    0.03     0.03 début
    0.33     0.30 mul
    0.54     0.21 sqr
    1.31     0.77 quomod
    1.94     0.63 quo
    2.60     0.66 sqrt
    9.81     7.21 gcd
   20.63    10.82 gcd_ex
> exemples/c/chrono-g -all 1000000 -r 10
    0.01     0.01 début
    0.57     0.56 mul
    0.99     0.42 sqr
    3.20     2.21 quomod
    5.40     2.20 quo
    7.06     1.66 sqrt
   60.82    53.76 gcd
  176.33   115.51 gcd_ex
>
```

Donc sur la machine de test (Pentium-4-2.8Ghz) avec le module `Slong`, une multiplication entre deux nombres d'un million de bits prend 30 millisecondes, le calcul du carré d'un nombre d'un million de bits prend 21 millisecondes, etc. Le second test présente les temps correspondants pour la bibliothèque `GMP`-4.1.4 sur la même machine.

6.4.2 digits

Détermine la plus petite puissance d'un nombre a dont l'écriture décimale commence par une suite donnée de chiffres (interface Ocaml uniquement). Formellement, on recherche un couple (x, y) d'entiers naturels minimal tel que $c < a^x/10^y < c + 1$ où c est le nombre désigné par la suite de chiffres voulue. La recherche est conduite avec des approximations à n bits de $\ln(a)$, $\ln(10)$ $\ln(c)$ et $\ln(c+1)$ où n est déterminé en fonction de a et c. Si elle échoue ou si la solution trouvée n'est pas reconnue comme minimale alors le calcul est relancé en doublant n. Les paramètres en ligne de commande sont dans cet ordre : la base a, la suite de chiffres c, et le nombre maximum d'essais à effectuer.

```
> exemples/ocaml/digits 3 1234567890 1
```

```
5399108054 2576029200
> exemples/ocaml/digits 3 1234567890 2
2440080224 1164214129 (minimal)
>
```

Donc $3^{5399108054} \approx 1234567890 \times 10^{2576029200}$, solution trouvée au premier essai, et $3^{2440080224} \approx 1234567890 \times 10^{1164214129}$, solution trouvée au deuxième essai. La deuxième solution est minimale.

6.4.3 pi

Calcul des n premières décimales de π (interfaces C, Caml, 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. Indiquez sur la ligne de commande le nombre n et les options de calcul :

`-d` affiche le détail des étapes et le temps de calcul de chaque étape.
`-noprint` n'effectue pas la conversion en chaîne du nombre calculé.
`-skip` effectue la conversion en chaîne, mais n'affiche que le début et la fin de la chaîne obtenue.
`-gcd` met la fraction obtenue après sommation de la série sous forme irréductible avant d'effectuer la division (cette mise sous forme irréductible est déconseillée, elle consomme plus de temps qu'elle n'en fait gagner dans la division).

```
> exemples/caml/pi-s 1000000 -d -skip
    0.00     0.00 module = Slong
    0.05     0.05 puiss-5
    0.41     0.36 sqrt
    3.04     2.63 série lb=6875847
    3.52     0.48 quotient
    4.29     0.77 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

>
```

6.4.4 shanks

Calcule la racine carrée b d'un nombre a modulo un nombre premier impair p (interfaces C, Caml, Ocaml et Pascal). Indiquez sur la ligne de commande les valeurs de a et p au moyen d'options `-p` *valeur* et `-a` *valeur*. Si l'une ou l'autre de ces valeurs n'est pas spécifiée, alors elle est tirée au hasard. Dans ce cas, l'option `-bits` *bits* indique combien de bits prendre pour le tirage au hasard.

```
> exemples/pascal/shanks-s -bits 200
p = 1005766304904354230760358867719456972987081899952626048942177
```

```
a = 97058061405060373035975326523959076688243798071458588343903 9
b = 15971954711903990910313854584615321009395927491416641891449 6
```

6.4.5 simple

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

6.4.6 sqrt-163

Calcule le nombre $\lfloor 10^n e^{\pi\sqrt{163}} \rfloor$ où n est donné sur la ligne de commande (interface Ocaml uniquement).

```
> exemples/ocaml/sqrt-163-opt 10
262537412640768743.9999999999
```

Noter que le résultat affiché suffit à prouver que $e^{\pi\sqrt{163}}$ n'est pas entier : s'il y avait une infinité de `9` après ceux affichés alors le programme n'aurait pas pu déterminer la partie entière demandée.

6.4.7 cmp, rcheck

Ces programmes ne sont disponibles qu'avec l'interface Ocaml. `cmp` effectue une série d'opérations tirées au hasard sur des opérandes entiers tirés au hasard, afin de détecter des bogues internes à `Numerix`. Deux modules de grands entiers doivent être sélectionnés en ligne de commande de façon à permettre la comparaison des résultats obtenus dans chaque module. Les autres options qui peuvent être passées en ligne de commande sont :

`-n` *bits* indique la taille en bits des opérandes ;
`-op` *opération* indique une opération particulière à tester ;
`-r` *compte* indique le nombre d'essais à effectuer ;
`-s` *seed* donne une graine pour le générateur aléatoire.

```
> exemples/ocaml/cmp -n 1000 -r 10000 -e clong -e gmp
Cmp(Clong,Gmp)
i=10000
>
```

10000 opérations effectuées sans détecter d'erreur.

`rcheck` est l'analogue de `cmp` pour les fonctions à valeurs réelles implémentées dans le module `Rfuns`. Le programme effectue une série de calculs pour chacune de ces fonctions et affiche sur le canal de sortie standard des instructions conformes à la syntaxe de `MuPAD` permettant de vérifier les résultats. Les options en ligne de commande sont :

`-bits` p indique le nombre de bits à prendre pour les opérandes a et b ;
`-n` n indique la précision à transmettre aux fonctions `xxx` de `Rfuns` ;
`-c` c indique le facteur multiplicatif à transmettre aux fonctions `r_xxx` ;

`-niter` i indique le nombre d'essais à effectuer pour chaque fonction ;
`-seed` s donne une graine pour le générateur aléatoire.

```
> exemples/ocaml/rcheck -niter 100 -bits 200 -c 1000000000000 | mupad -P pe

   *----*    MuPAD 2.5.3 -- The Open Computer Algebra System
  /|   /|
 *----* |    Copyright (c)  1997 - 2003  by SciFace Software
 | *--|-*                   All rights reserved.
 |/   |/
 *----*      Licensed to:   Michel Quercia


                              c = 1000000000000

                                   x = 1

     u = 145237906349845897235553079725126760966964128040718229912093 1

     v = -11946610261415842471497825548686767812337415664907660430492

                                  f = exp

                                 r = ceil
>
```

Une seule erreur a été détectée : `Numerix` retourne le résultat $x = 1$ comme valeur de $\lceil c \times \exp(u/v) \rceil$ tandis que MuPAD trouve une autre valeur (non affichée). Après vérification il se trouve que c'est MuPAD qui tort, le résultat de `Numerix` est correct.