Interface for module Creal

1. Constructive reals are implemented by the following abstract datatype t. If x is a constructive real, then the function call *approx* x n returns an approximation of x up to 4^{-n}, as an arbitrary precision integer x_n such that $|4^n \cdot x - x_n| < 1$.

open Gmp

type t

val $approx$: $t \rightarrow int \rightarrow Z.t$

val msd : $t \rightarrow int$

2. Basic operations.

val add : $t \rightarrow t \rightarrow t$
val neg : $t \rightarrow t$
val sub : $t \rightarrow t \rightarrow t$

val abs : $t \rightarrow t$

val mul : $t \rightarrow t \rightarrow t$
val inv : $t \rightarrow t$
val div : $t \rightarrow t \rightarrow t$

val pow_int : $t \rightarrow int \rightarrow t$

val $sqrt$: $t \rightarrow t$

3. Transcendental functions. *log* x y is $\log_x(y)$.

val ln : $t \rightarrow t$
val log : $t \rightarrow t \rightarrow t$

val exp : $t \rightarrow t$
val pow : $t \rightarrow t \rightarrow t$

4. Trigonometric functions.

val sin : $t \rightarrow t$
val cos : $t \rightarrow t$
val tan : $t \rightarrow t$

val $arcsin$: $t \rightarrow t$
val $arccos$: $t \rightarrow t$
val $arctan$: $t \rightarrow t$

5. *arctan_reciproqual* n is $\arctan(1/n)$, but is more efficient than using *arctan*.

val *arctan_reciproqual* : *int* → *t*

6. Hyperbolic functions.

val *sinh* : *t* → *t*
val *cosh* : *t* → *t*
val *tanh* : *t* → *t*

val *arcsinh* : *t* → *t*
val *arccosh* : *t* → *t*
val *arctanh* : *t* → *t*

7. Some constants.

val *zero* : *t*
val *one* : *t*
val *two* : *t*

val *pi* : *t*
val *pi_over_2* : *t*

val *e* : *t*

8. Comparisons. *cmp* is absolute comparison: it may not terminate and only returns -1 or +1. *rel_cmp* is relative comparison, up to 4^{-k}, and it returns -1, 0 or +1.

val *cmp* : *t* → *t* → *int*
val *rel_cmp* : *int* → *t* → *t* → *int*

val *min* : *t* → *t* → *t*
val *max* : *t* → *t* → *t*

9. Coercions. *to_q* and *to_float* expect a precision. *to_float* x n returns the best floating point representation of the rational $x_n/4^n$. *of_string* expects a base as second argument.

val *of_int* : *int* → *t*
val *of_z* : *Z.t* → *t*
val *of_q* : *Q.t* → *t*
val *of_float* : *float* → *t*
val *of_string* : *string* → *int* → *t*

val *to_float* : *t* → *int* → *float*
val *to_q* : *t* → *int* → *Q.t*

10. Coercion to type *string*. Given a decimal precision p, *to_string* x p returns a decimal approximation d of x with either p digits such that $|d - x| < 10^{-p}$, or $p + 1$ digits such that $|d - x| < 10^{-p-1}$.

to_beautiful_string returns the same decimal number but with digits packed 5 by 5.

```
val to_string : t → int → string
val to_beautiful_string : t → int → string
```

11. Infix notations.

```
module Infixes : sig
  val ( +! ) : t → t → t
  val ( −! ) : t → t → t
  val ( ∗! ) : t → t → t
  val ( /! ) : t → t → t
end
```

Module Creal

12. This module implements constructive reals (exact real numbers), following the algorithms given in Valérie Ménissier-Morain's thesis (`http://www-calfor.lip6.fr/~vmm/`). In the following, pages refer to this document.

13. Representation. A constructive real is represented by an approximation function (field *approximate*). If x is a real number, its approximation function applied to an integer n (in type *int*) returns an integer x_n (in type $Z.t$) such that $|4^n \cdot x - x_n| < 1$.

For efficiency, we add a field *cache* to keep the best approximation computed so far. (Notice that it is safe to use type *int* for the number of digits, since an integer with a number of digits exceeding the capacity of machine integers would exceed the memory capacity.)

The field *msd* is a cache for the most significant digit (see section 21 below).

```
type t = {
  mutable cache : (int × Z.t) option;
  mutable msd : int option;
  approximate : int → Z.t }

let create f = { cache = None; msd = None; approximate = f }
```

14. Computing the approximation of x to precision n is easy: either we have already computed a better approximation and the result is just a "shift" of that value (Property 6 page 46), or we compute *x.approximate n* and we cache its result before returning it.

```
let fdiv_Bexp z n =
  if n ≡ 0 then
    z
  else if n > 0 then
    Z.fdiv_q_2exp z (n + n)
  else
    Z.mul2exp z (−(n + n))

let approx x n =
  let compute () =
    let z = x.approximate n in x.cache ← Some (n, z); z
  in
  match x.cache with
    | None → compute ()
    | Some (m, a) → if n ≤ m then fdiv_Bexp a (m − n) else compute ()
```

15. Some useful constants in $Z.t$ and $Q.t$.

```
let z_zero = Z.from_int 0
let z_one = Z.from_int 1
let z_two = Z.from_int 2
let z_three = Z.from_int 3
let z_four = Z.from_int 4

let q_half = Q.from_ints 1 2
let q_zero = Q.from_ints 0 1
let q_one = Q.from_ints 1 1
let q_two = Q.from_ints 2 1
let q_four = Q.from_ints 4 1
```

16. Utility functions over $Z.t$ and $Q.t$.

```
let z_gt x y = Z.cmp x y > 0
let z_le x y = Z.cmp x y ≤ 0

let z_between lb x up = z_le lb x ∧ z_le x up

let z_even x = (Z.cmp (Z.cdiv_r_ui x 2) z_zero) ≡ 0

let q_max q1 q2 = if Q.cmp q1 q2 ≥ 0 then q1 else q2

let q_abs q = if Q.sgn q < 0 then Q.neg q else q
```

17. Roundings. Floor, ceil and Gauß rounding over $Q.t$. The Gauß rounding of x, written $\lfloor x \rceil$, is the (only) integer such that $\lfloor x \rceil - \frac{1}{2} \leq x < \lfloor x \rceil + \frac{1}{2}$.

```
let q_floor q = Z.fdiv_q (Q.get_num q) (Q.get_den q)

let q_ceil q = Z.cdiv_q (Q.get_num q) (Q.get_den q)
```

```
let gauss_round q =
  let q′ = Q.add q q_half in
  Z.fdiv_q (Q.get_num q′) (Q.get_den q′)
```

```
let gauss_round_z_over_4 z = Z.fdiv_q_2exp (Z.add_ui z 2) 2
```

18. Addition (Algorithm 2 page 50). We have $(x + y)_n = \lfloor (x_{n+1} + y_{n+1})/4 \rceil$. We do not try to cache a value for $x + y$ given the cached values for x and y, if any, since it may require some computation (some shifts). Moreover, this is exactly what will be done by the first call to *approx* on $x + y$ if the precision asked is less than $min(x, y) - 2$.

```
let add x y =
  create
    (function n →
       let sn = succ n in
       gauss_round_z_over_4 (Z.add (approx x sn) (approx y sn)))
```

```
let (+!) = add
```

19. Negation is immediate and subtraction is the composition of addition and negation (Algorithm 3 page 51). The cached value for x is immediatly cached in -x (at no cost).

```
let cache_neg = function
  | None → None
  | Some (n, a) → Some (n, Z.neg a)
```

```
let neg x =
  { cache = cache_neg x.cache;
    msd = x.msd;
    approximate = function n → Z.neg (approx x n) }
```

```
let sub x y = x +! (neg y)
```

```
let (−!) = sub
```

20. Absolute value.

```
let abs x = create (function n → Z.abs (approx x n))
```

21. Most significant digit (*msd*, Definition 9 page 47). It is defined by

$$\mathrm{msd}(x) = \min \{n \in Z \mid |x_n| > 1\}$$

Note that it does not terminate in 0.

```
let compute_msd x =
  let rec look_up n =
    (* |x_{n-1}| ≤ 1 *)
    let xn = Z.abs (approx x n) in
    if z_gt xn z_one then n else look_up (succ n)
  and look_down n =
    (* |x_{n+1}| > 1 *)
    let xn = Z.abs (approx x n) in
    if z_gt xn z_one then look_down (pred n) else succ n
  in
  let x0 = Z.abs (approx x 0) in
  if z_gt x0 z_one then look_down (−1) else look_up 1

let msd x = match x.msd with
  | None → let m = compute_msd x in x.msd ← Some m; m
  | Some m → m
```

22. Version of *msd* with a maximal bound on the iteration process (used in function *mul* to avoid non-termination when multiplicating by 0).

```
let msd_with_max m x =
  let rec look_up n =
    if n ≥ m then n else
    let xn = Z.abs (approx x n) in
    if z_gt xn z_one then n else look_up (succ n)
  and look_down n =
    let xn = Z.abs (approx x n) in
    if z_gt xn z_one then look_down (pred n) else succ n
  in
  let x0 = Z.abs (approx x 0) in
  if z_gt x0 z_one then look_down (−1) else look_up 1
```

23. *mul_Bexp* and *div_Bexp* respectively multiplies and divides an integer by B^n (works whatever the sign of n is). The result is a rational.

```
let mul_Bexp z n =
  if n ≡ 0 then
    Q.from_z z
  else if n > 0 then
    Q.from_z (Z.mul2exp z (n + n))
  else
    Q.from_zs z (Z.mul2exp z (−(n + n)))

let bexp n = mul_Bexp z_one n
```

```
let div_Bexp z n =
  if n ≡ 0 then
    Q.from_z z
  else if n > 0 then
    Q.from_zs z (Z.mul2exp z_one (n + n))
  else
    Q.from_z (Z.mul2exp z (−(n + n)))
```

24. Multiplication (Algorithm 4 page 51).

```
let mul x y =
  create
    (function n →
       let d = (n + 2) / 2 in
       let msd′ = msd_with_max (n + 3 − d) in
       let px = max (n − (msd′ y) + 3) d
       and py = max (n − (msd′ x) + 3) d in
       let xpx = approx x px
       and ypy = approx y py in
       let z = gauss_round (div_Bexp (Z.add_ui (Z.abs (Z.mul xpx ypy)) 1)
                                     (px + py − n)) in
       if Z.sgn xpx = Z.sgn ypy then z else Z.neg z)

let ( *! ) = mul
```

25. Inverse (Algorithm 5 page 53) and division.

```
let inv x =
   create
     (function n →
        let msdx = msd x in
        if n ≤ − msdx then
          z_zero
        else
          let k = n + 2 × msdx + 1 in
          let xk = approx x k in
          let q = Q.div (bexp (k + n)) (Q.from_z xk) in
          if z_gt xk z_one then q_ceil q else q_floor q)

let div x y = x *! (inv y)

let (/!) = div
```

26. Square root (Algorithm 6 page 56).

```
let sqrt x =
  create
    (function n →
       let x2n = approx x (n + n) in
       if Z.sgn x2n < 0 then invalid_arg "Creal.sqrt";
       Z.sqrt x2n)
```

27. Coercions from integers and rationals (Algorithm 1 page 49) and coercion to rationals.

```
let fmul_Bexp q n =
  if n ≡ 0 then
    q_floor q
  else if n > 0 then
    Z.fdiv_q (Z.mul2exp (Q.get_num q) (n + n)) (Q.get_den q)
  else
    q_floor (Q.div q (Q.from_z (Z.mul2exp z_one (−(n + n)))))

let of_z z =
  { cache = Some (0, z);
    msd = None;
    approximate = function n → fmul_Bexp (Q.from_z z) n }

let of_q q = create (fmul_Bexp q)

let to_q x n =
  let xn = approx x n in
  Q.div (Q.from_z xn) (bexp n)

let of_int n = of_z (Z.from_int n)

let zero = of_int 0
let one = of_int 1
let two = of_int 2
let four = of_int 4
```

28. Power of a real to a small integer.

```
let rec pow_int x n =
  if n ≡ 0 then
    one
  else if n < 0 then
    inv (pow_int x (−n))
  else
    let y = pow_int (mul x x) (n / 2) in
    if n mod 2 ≡ 0 then y else mul y x
```

```
let rec pow_z x n =
  let c = Z.cmp_si n 0 in
  if c ≡ 0 then
    one
  else if c < 0 then
    inv (pow_z x (Z.neg n))
  else
    let y = pow_z (mul x x) (Z.fdiv_q_2exp n 1) in
    if Z.cmp_si (Z.dmod_ui n 2) 0 ≡ 0 then y else mul y x
```

29. Alternate power series. The following function *alternate_powerserie_* computes B^pS where S is the partial sum of an alternate power serie such that the remainder is less than B^{-p}, that is $S = \sum_{i=0}^{i=n}(-1)^i a_i$ with $a_{n+1} < B^{-p}$. The alternate power serie is given by its first term a_0 and a function *next* such that $a_{n+1} = next\ n\ a_n$.

```
let alternate_powerserie_ a0 next p =
  let eps = bexp (−p) in
  let rec sum s n an =
    (∗ s is already the sum up to a_n ∗)
    let asn = next n an in
    if Q.cmp (q_abs asn) eps < 0 then
      s
    else
      sum (if n mod 2 ≡ 0 then Q.sub s asn else Q.add s asn) (n + 1) asn
  in
  Q.div (sum a0 0 a0) eps
```

30. A specialized function to compute $atan(1/m)$ where m is a small integer.

```
let arctan_reciproqual m =
  let m_inverse = Q.from_ints 1 m in
  let m_inverse_square = Q.mul m_inverse m_inverse in
  create
    (fun n →
       let eps = bexp (−n) in
       let rec sum s sign k p =
         (∗ s is already the sum up to a_k ∗)
         let p′ = Q.mul p m_inverse_square in
         let t = Q.mul p′ (Q.from_ints 1 (k + 2)) in
         if Q.cmp t eps < 0 then
           s
         else
           sum (if sign then Q.add s t else Q.sub s t) (¬ sign) (k + 2) p′
       in
       fmul_Bexp (sum m_inverse false 1 m_inverse) n)
```

31. π is defined using *arctan*, with the well-known formula (Algorithm 13 page 68)

$$\frac{\pi}{4} = 12\arctan\left(\frac{1}{18}\right) + 8\arctan\left(\frac{1}{57}\right) - 5\arctan\left(\frac{1}{239}\right)$$

```
let pi =
    (of_int 48 *! arctan_reciproqual 18)
  +! (of_int 32 *! arctan_reciproqual 57)
  -! (of_int 20 *! arctan_reciproqual 239)

let pi_over_2 = pi /! two
```

32. Arctangent (Algorithm 12 page 64).

```
let arctan_ x =
  let square_x = Q.mul x x in
  let next n an =
    Q.mul (Q.mul an square_x) (Q.from_ints (2 × n + 1) (2 × n + 3))
  in
  alternate_powerserie_ x next

let arctan_def x =
  create
    (function n →
      let k = max 0 (n + 1) in
      let xk = approx x k in
      if Z.cmp_si xk 0 ≡ 0 then
        z_zero
      else
        let q = Q.from_zs xk (Z.pow_ui_ui 4 k) in
        q_floor (Q.add
                   (Q.div (Q.add (arctan_ q (n + 1)) q_one) q_four)
                   (Q.div
                      (bexp (n + k))
                      (Q.add (bexp (2 × n + 2))
                         (Q.from_z (Z.add (Z.mul xk xk) xk))))))
```

33. The above definition of *arctan* converges very slowly when $|x| \geq 1$. The convergence is accelerated using the following identities:

$$\begin{array}{lll} \arctan(x) & = -\pi/2 - \arctan(1/x) & \text{when } x < -1 \\ & = -\pi/4 - \arctan((1-x^2)/(2x))/2 & \text{when } x \approx -1 \\ & = +\pi/4 - \arctan((1-x^2)/(2x))/2 & \text{when } x \approx 1 \\ & = +\pi/2 - \arctan(1/x) & \text{when } x > 1 \end{array}$$

We use the approximation of x at order 1 to discriminate between the cases.

```
let arctan x =
  let x1 = approx x 1 in
  if Z.cmp_si x1 (−5) < 0 then
    (∗ x < −1 ∗)
    neg (pi_over_2 +! arctan_def (inv x))
  else if Z.cmp_si x1 (−3) ≤ 0 then
    (∗ x close to −1 ∗)
    neg (pi_over_2 +! arctan_def ((one −! x ∗! x) /! (two ∗! x))) /! two
  else if Z.cmp_si x1 5 > 0 then
    (∗ x > 1 ∗)
    pi_over_2 −! arctan_def (inv x)
  else if Z.cmp_si x1 3 ≥ 0 then
    (∗ x close to 1 ∗)
    (pi_over_2 −! arctan_def ((one −! x ∗! x) /! (two ∗! x))) /! two
  else
    (∗ x close to 0 ∗)
    arctan_def x
```

34. Arcsinus and arccosinus are derived from arctangent (Algorithm 14 page 69). We use $\arcsin(x) + \arccos(x) = \pi/2$ to avoid non-termination of arcsin(1) and arccos(0).

```
let arcsin_def x = arctan (x /! (sqrt (one −! (x ∗! x))))

let arccos_def x = arctan ((sqrt (one −! (x ∗! x))) /! x)

let arcsin x =
  let x1 = approx x 1 in
  if z_le (Z.abs x1) z_two then (∗ —x— ¡ 3/4 ∗) arcsin_def x
  else if z_le z_three x1 then (∗ x ¿ 1/2 ∗) pi_over_2 −! arccos_def x
  else (∗ x ¡ -1/2 ∗) arccos_def (neg x) −! pi_over_2

let arccos x =
  let x1 = approx x 1 in
  if z_le (Z.abs x1) z_two then (∗ —x— ¡ 3/4 ∗) pi_over_2 −! arcsin_def x
  else if z_le z_three x1 then (∗ x ¿ 1/2 ∗)arccos_def x
  else (∗ x ¡ -1/2 ∗) pi −! arccos_def (neg x)
```

35. Sinus (Algorithm 15 page 69).

```
let rec sin_ x p =
  if Q.cmp x q_zero ≥ 0 then
    let square_x = Q.mul x x in
    let next n an =
      Q.mul (Q.mul (Q.mul an square_x) (Q.from_ints 1 (2 × n + 2)))
            (Q.from_ints 1 (2 × n + 3))
    in
```

```
    alternate_powerserie_ x next p
  else
    Q.neg (sin_ (Q.neg x) p)

let sin x =
  let p = Z.sub_ui (approx (x /! pi) 0) 1 in
  let theta = if Z.cmp_si p 0 ≡ 0 then x else x −! ((of_z p) ∗! pi) in
  let z = pi_over_2 in
  create
    (fun n →
       let k = max 2 (n + 2) in
       let zk = approx z k in
       let twozk = Z.mul2exp zk 1 in
       let threezk = Z.mul_ui zk 3 in
       let fourzk = Z.mul2exp zk 2 in
       let thetak = approx theta k in
       if (z_between z_zero thetak z_one) ∨
          (z_between (Z.sub_ui fourzk 4) thetak (Z.add_ui fourzk 4)) ∨
          (z_between (Z.sub_ui twozk 2) thetak (Z.add_ui twozk 2)) then
         z_zero
       else if z_between (Z.sub_ui zk 1) thetak (Z.add_ui zk 1) then
         let bn = Z.mul2exp z_one (n + n) in
         if z_even p then bn else Z.neg bn
       else if z_between (Z.sub_ui threezk 3) thetak (Z.add_ui threezk 3) then
         let bn = Z.mul2exp z_one (n + n) in
         if z_even p then Z.neg bn else bn
       else
         let q = Q.from_zs thetak (Z.pow_ui_ui 4 k) in
         let s = sin_ q (n + 2) in
         let bw = Q.from_ints 16 1 in
         let bn_k = bexp (n − k) in
         let r =
           if (z_between z_two thetak (Z.sub_ui zk 2)) ∨
              (z_between (Z.add_ui zk 2) thetak (Z.sub_ui twozk 3)) then
             q_floor (Q.add (Q.div (Q.add s q_one) bw) bn_k)
           else
             q_ceil (Q.sub (Q.div (Q.sub s q_one) bw) bn_k)
         in
         if z_even p then r else Z.neg r)
```

36. Cosinus and tangent are derived from sinus (Algorithm 16 page 78).

```
let cos x = sin (pi_over_2 −! x)

let tan x = (sin x) /! (cos x)
```

37. Euler constant e.

```
type sum_cache = {
  mutable order : int;
  mutable sum : Q.t; (* sum up to order *)
  mutable term : Q.t; (* last term a_order *)
  mutable prec : int
}

let e =
  let e_cache = { order = 1; sum = q_two; term = q_one; prec = 0 } in
  create
    (fun p →
       if p ≤ e_cache.prec then
         fmul_Bexp e_cache.sum p
       else
         let eps = bexp (−p) in
         let rec sum s n an =
           let rn = Q.mul (Q.from_ints 1 n) an in
           if Q.cmp rn eps ≤ 0 then begin
             e_cache.order ← n;
             e_cache.sum ← s;
             e_cache.term ← an;
             e_cache.prec ← p;
             fmul_Bexp s p
           end else
             let asn = Q.mul (Q.from_ints 1 (n + 1)) an in
             sum (Q.add s asn) (n + 1) asn
         in
         sum e_cache.sum e_cache.order e_cache.term)
```

38. Natural logarithm (Algorithm 9 page 62).

```
let ln_above_1 r =
  let y = Q.div (Q.sub r q_one) (Q.add r q_one) in
  let y_square = Q.mul y y in
  let one_minus_y_square = Q.sub q_one y_square in
  fun n →
    let eps = bexp (−n) in
    let rec sum s k p =
      (* s is already the sum up to a_k *)
      let p′ = Q.mul p y_square in
      let t = Q.mul p′ (Q.from_ints 1 (k + 2)) in
      if Q.cmp (Q.div t one_minus_y_square) eps < 0 then
        Q.mul q_two s
```

```
      else
        sum (Q.add s t) (k + 2) p′
    in
    Q.div (sum y 1 y) eps

let rec ln_ r =
  if Q.cmp r q_zero ≤ 0 then invalid_arg "Creal.ln";
  let cmp1 = Q.cmp r q_one in
  if cmp1 < 0 then
    (∗ r < 1 ∗)
    let ln_inverse_r = ln_ (Q.inv r) in
    (fun n → Q.neg (ln_inverse_r n))
  else if cmp1 ≡ 0 then
    (∗ r = 1 ∗)
    (fun n → q_zero)
  else
    (∗ r > 1 ∗)
    ln_above_1 r

let ln_4 = let f = ln_above_1 q_four in create (fun n → q_floor (f n))

let rec ln x =
  let msd_x = msd x in
  let k = − msd_x + 1 in
  if k ≢ 0 then
    ln (x /! (of_q (bexp k))) +! (of_int k) ∗! ln_4
  else
    create
      (fun n →
         let w = 2 − min 0 n in
         let k = n + msd_x + w in
         let xk = Q.from_z (approx x k) in
         let q = Q.div xk (bexp k) in
         q_floor (Q.add (Q.div (Q.add (ln_ q (n + w)) q_one) (bexp w))
                        (Q.div (bexp n) xk)))

let log x y = ln y /! ln x
```

39. Inverses of hyperbolic functions.

```
let arcsinh x = ln (x +! sqrt (x ∗! x +! one))
let arccosh x = ln (x +! sqrt (x ∗! x −! one))
let arctanh x = ln ((one +! x) /! (one −! x)) /! two
```

40. Exponential (Algorithm 7 page 57).

```
let exp_neg_ r =
  (* −1 ≤ r < 0 *)
  let r = q_abs r in
  let next n an = Q.mul (Q.mul an r) (Q.from_ints 1 (n + 1)) in
  create (fun n → q_floor (alternate_powerserie_ q_one next n))

let exp_ r =
  if Q.cmp r q_zero ≡ 0 then
    one
  else
    let s_floor_r = Z.add_ui (q_floor r) 1 in
    mul (pow_z e s_floor_r) (exp_neg_ (Q.sub r (Q.from_z s_floor_r)))

let exp x =
  create
    (fun n →
       let qbn = bexp n in
       let bn = of_q qbn in
       let invqbn = Q.inv qbn in
       let one_plus_invqbn = Q.add q_one invqbn in
       let test1 () =
         let lsup = log four (of_int 7 /! ln ((bn +! one) /! (bn −! one))) in
         let l = Z.int_from (approx lsup 0) + 1 in
         let xl = approx x l in
         let log1 = q_floor (ln_ (Q.sub q_one invqbn) l) in
         let log2 = q_floor (ln_ one_plus_invqbn l) in
         (Z.cmp (Z.add log1 z_two) xl < 0) ∧
         (Z.cmp xl (Z.sub log2 z_two) < 0)
       in
       let test2 () =
         let x0 = approx x 0 in
         let m = Z.sub (q_floor (ln_ one_plus_invqbn 0)) z_two in
         Z.cmp x0 m ≤ 0
       in
       if (n > 0 ∧ test1 ()) ∨ (n ≤ 0 ∧ test2 ()) then
         fmul_Bexp q_one n
       else
         let msd_x = msd x in
         let clogBe =
           if Z.cmp (approx x msd_x) z_one ≥ 0 then
             Q.from_ints 577080 100000
           else
             Q.from_ints (−72134) 100000
         in
```

```
          let d2 = Q.div clogBe (bexp msd_x) in
          let p = max 0 (n + 1) in
          let d = q_max (Q.from_ints (−p) 1) d2 in
          let k2 = q_ceil (Q.add d (Q.from_ints 44732 100000)) in
          let k = max 1 (max msd_x (p + 1 + Z.int_from k2)) in
          let bk = bexp k in
          let xk = approx x k in
          let xkBk = Q.div (Q.from_z xk) bk in
          let exp_xkBk_p = approx (exp_ xkBk) p in
          if Z.cmp exp_xkBk_p z_zero ≤ 0 then
            z_zero
          else
            q_ceil (Q.mul (Q.sub (Q.div (Q.from_z exp_xkBk_p) q_four) q_one)
                          (Q.sub q_one (Q.inv bk))))

let pow x y = exp (y *! ln x)
```

41. Hyperbolic functions.

```
let sinh x = (exp x −! exp (neg x)) /! two

let cosh x = (exp x +! exp (neg x)) /! two

let tanh x = sinh x /! cosh x
```

42. Comparisons. *cmp* is absolute comparison and *rel_cmp* is comparison up to 4^{-k}.

```
let cmp x y =
  let rec cmp_rec n =
    let xn = approx x n in
    let yn = approx y n in
    if z_gt (Z.add_ui xn 1) (Z.sub_ui yn 1) ∧
        z_gt (Z.add_ui yn 1) (Z.sub_ui xn 1)
    then
      cmp_rec (succ n)
    else
      if z_le (Z.add_ui xn 1) (Z.sub_ui yn 1) then -1 else 1
  in
  cmp_rec 0

let rel_cmp k x y =
  let rec cmp_rec n =
    let xn = approx x n in
    let yn = approx y n in
    if z_gt (Z.add_ui xn 1) (Z.sub_ui yn 1) ∧
        z_gt (Z.add_ui yn 1) (Z.sub_ui xn 1) ∧ n ≤ k + 2
    then
```

```
      cmp_rec (succ n)
    else if z_le (Z.add_ui xn 1) (Z.sub_ui yn 1) then
      -1
    else if z_le (Z.add_ui yn 1) (Z.sub_ui xn 1) then
      1
    else
      0
  in
  cmp_rec 0
```

43. Coercions to and from type *float*.

```
let of_float f = of_q (Q.from_float f)
```

```
let to_float x n = Q.float_from (to_q x n)
```

44. Coercion to and from type *string*.

```
let of_string s base =
  try
    begin
      try
        let n = String.length s in
        let p = String.index s '.' in
        let dec = n − p − 1 in
        let s′ = (String.sub s 0 p) ^ (String.sub s (p + 1) dec) in
        of_q (Q.from_zs (Z.from_string_base base s′) (Z.pow_ui_ui base dec))
      with Not_found →
        of_z (Z.from_string_base base s)
    end
  with Invalid_argument _ → invalid_arg "Creal.of_string"
```

45. Decimal approximation of x at order p. We look for an integer n such that $|10^p x - n| < 1/2$ i.e. the integer closest to $10^p x$. There is sometimes no such integer but then we can find a decimal approximation at order $p+1$.

We first compute $y = 10^p x$ and *approx* y 3 i.e. an approximation $y_3/64$ of y. Let q and r be the quotient and remainder of the division $y_3/64$ such that $y_3 = 64q + r$ and $0 \leq r < 63$. If $r \leq 31$ then n is q; If $r \geq 33$ then n is $q+1$; Otherwise $10q+5$ is a decimal approximation of x at order $p+1$.

```
let to_string_aux x p =
  if p < 0 then invalid_arg "Creal.to_string";
  let tenp = Z.pow_ui_ui 10 p in
  let y = mul (of_z tenp) x in
  let y3 = approx y 3 in
  let q, r = Z.fdiv_qr_ui y3 64 in
```

```
  let r = Z.int_from r in
  let n, p =
    if r ≤ 31 then q, p
    else if r ≥ 33 then Z.add_ui q 1, p
    else Z.add_ui (Z.mul_ui q 10) 5, succ p
  in
  let ns = Z.string_from (Z.abs n) in
  let lns = String.length ns in
  let ins, dns =
    if lns ≥ p + 1 then
      String.sub ns 0 (lns − p), String.sub ns (lns − p) p
    else
      "0", String.make (p − lns) '0' ^ ns
  in
  Z.sgn n, ins, dns

let to_string x p =
  let sgn, i, f = to_string_aux x p in
  (if sgn < 0 then "-" else "") ^ i ^ "." ^ f
```

46. Coercion to type *string* with digits packed 5 by 5.

```
let string_concat = String.concat ""

let beautiful s =
  let n = String.length s in
  let eol i = if (i + 5) mod 65 ≡ 0 then "\n" else "␣" in
  let rec cut i =
    String.sub s i (min 5 (n − i)) ::
    if i < n − 5 then eol i :: cut (i + 5) else []
  in
  string_concat (cut 0)

let to_beautiful_string x p =
  let sgn, i, f = to_string_aux x p in
  let nl = if String.length i + String.length f > 75 then "\n" else "" in
  (if sgn < 0 then "-" else "") ^ i ^ "." ^ nl ^ beautiful f
```

min and max here not to hide Pervasives's min and max
* let min x y = create (fun n -¿ Z.min (approx x n) (approx y n)) let max x y = create (fun n -¿ Z.max (approx x n) (approx y n)) *

```
let min x y =
  let min_xy = ref None in
  create
    (fun n → match !min_xy with
```

```
        | Some r → approx r n
        | None →
            let xn = approx x n in
            let yn = approx y n in
            if z_le (Z.add_ui xn 1) (Z.sub_ui yn 1) then begin
              min_xy := Some x; xn
            end else if z_le (Z.add_ui yn 1) (Z.sub_ui xn 1) then begin
              min_xy := Some y; yn
            end else Z.min xn yn)

let max x y =
  let max_xy = ref None in
  create
    (fun n → match !max_xy with
        | Some r → approx r n
        | None →
            let xn = approx x n in
            let yn = approx y n in
            if z_le (Z.add_ui xn 1) (Z.sub_ui yn 1) then begin
              max_xy := Some y; yn
            end else if z_le (Z.add_ui yn 1) (Z.sub_ui xn 1) then begin
              max_xy := Some x; xn
            end else Z.max xn yn)

module Infixes = struct
  let (+!) = add
  let (−!) = sub
  let ( *! ) = mul
  let (/!) = div
end
```