



HAL
open science

The CAML numbers reference manual

V. Menissier-Morain

► **To cite this version:**

V. Menissier-Morain. The CAML numbers reference manual. RT-0141, INRIA. 1992, pp.157. inria-00070027

HAL Id: inria-00070027

<https://inria.hal.science/inria-00070027>

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: +33(1)39 63 55 11

Rapports Techniques

N° 140

Programme 2

*Calcul symbolique, Programmation
et Génie logiciel*

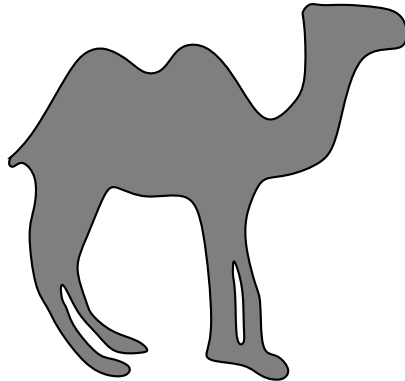
THE CAML NUMBERS REFERENCE MANUAL

Valérie MÉNISSIER-MORAIN

Juillet 1992

The CAML Numbers Reference Manual

Valérie Ménissier-Morain



Projet Formel

INRIA-ENS

Version 3.1

The CAML Numbers Reference Manual

Valérie MÉNISSIER-MORAIN¹

Abstract

This is the reference manual for the arithmetic of the functional language CAML V3.1. It is available by anonymous ftp from `ftp.inria.fr` as a compressed PostScript file `lang/caml/V3.1/doc_arith.tar.Z`. This arithmetic includes floating point numbers, arbitrary large integers and rationals, and a complete set of primitives. Its implementation relies on the efficient `BigNum` package [2].

Résumé

Ceci est le manuel de référence de l'arithmétique du langage fonctionnel CAML V3.1. Il est disponible sous forme de fichier PostScript compressé `lang/caml/V3.1/doc_arith.tar.Z` par ftp anonyme sur `ftp.inria.fr`. Cette arithmétique contient des nombres en virgule flottante, des entiers et rationnels de longueur arbitraire, et un ensemble complet de primitives. Cette implémentation s'appuie sur le package `BigNum` [2].

¹This work is partly supported by Allocation de Recherche de troisième Cycle of Ministère de la Recherche et de la Technologie (contrat 89553) and Institut National de Recherche en Informatique et Automatique.

A Jean-Louis Thomas

It makes me nervous to fly on airplanes since I know
they are designed using floating point arithmetic.

(Anston Householder)

Acknowledgements

I thank Gérard Huet for the interest he took first in the implementation and then in the documentation of this arithmetic.

While trying to write an alternative implementation of CAML, Damien Doligez and Pierre Weis asked me to implement a new arbitrary precision rational arithmetic for CAML using the BigNum Package.

This CAML implementation is still under development but Pierre Weis, the great meharist, asked me to insert this new arithmetic in the V3.1 CAML version as the main change. He believes in this new implementation and supported me all along this work.

This work would have been much harder without the constant help of François Morain and Bernard Serpette.

Thierry Coquand made me aware of the problem of normalization vs. non-normalization of rational numbers.

With my grateful thanks to Ian Jacobs for his \TeX -nical contribution to this documentation.

I thank the CAML Team and the Formel Project for the interest to my work and quality of the working and programming environment.

I am thankful to all of them for their technical contribution, their judicious advice and their friendly support.

And last but not least, I thank all the rereaders for their invaluable help to find and correct typos and other errors.

Table des matières

1	Introduction	15
1.1	Casual user	15
1.2	Regular user	18
2	Operations on numbers of type int	21
2.1	Arithmetic operations for <code>ints</code>	21
2.1.1	Incrementing and decrementing	21
2.1.2	Arithmetic operations for <code>ints</code>	23
2.2	Comparisons between <code>ints</code>	25
2.3	Coercion functions on <code>ints</code>	26
2.3.1	Coercion between type <code>float</code> and <code>int</code>	26
2.3.2	Coercion between characters and <code>ints</code>	27
2.3.3	Coercion between strings and <code>ints</code>	27
2.4	Logical operations	30
2.5	Miscellaneous functions on type <code>int</code>	31
3	Floating point numbers	33
3.1	Arithmetic operations for floating point numbers	33
3.1.1	Incrementing and decrementing	33
3.1.2	Arithmetic operations for floating point numbers	34
3.2	Comparisons between floating point numbers	34
3.3	Floating point transcendental operations	35
3.4	Coercion functions on floating point numbers	36
3.4.1	Reading floating point numbers	36
4	Natural numbers	39
4.1	Natural numbers representation	39
4.2	Creation of natural numbers	40
4.3	Assignment of natural numbers	40
4.4	Lengths of a natural number	42
4.5	Miscellaneous informations on natural numbers	42
4.6	Arithmetic operations on natural numbers	44
4.6.1	Addition operators on natural numbers	44
4.6.2	Subtraction operators on natural numbers	47
4.6.3	Multiplication operators on natural numbers	49

4.6.4	Division operators on natural numbers	54
4.7	Comparisons on natural numbers	57
4.8	Coercion functions on natural numbers	59
4.8.1	Coercion between types <code>int</code> and <code>nat</code>	59
4.8.2	Coercion between types <code>float</code> and <code>nat</code>	60
4.8.3	Coercion between types <code>string</code> and <code>nat</code>	60
4.9	Logical operations on natural numbers	65
4.10	Miscellaneous power functions	66
5	Big integers	69
5.1	Creation of a <code>big_int</code>	69
5.2	Arithmetic operations for <code>big_ints</code>	70
5.2.1	Incrementing <code>big_ints</code>	70
5.2.2	Arithmetic operations for <code>big_ints</code>	70
5.3	Comparisons between <code>big_ints</code>	71
5.4	Coercion functions on <code>big_ints</code>	72
5.4.1	Coercion between <code>ints</code> and <code>big_ints</code>	72
5.4.2	Coercion between floating point numbers and <code>big_int</code>	73
5.4.3	Coercion between <code>nats</code> and <code>big_ints</code>	73
5.4.4	Coercion between strings and <code>big_int</code>	74
5.5	Miscellaneous information functions on <code>big_ints</code>	78
5.6	Miscellaneous power functions	78
6	Rational numbers of type <code>ratio</code>	81
6.1	Normalization of <code>ratios</code>	81
6.2	Creation of a <code>ratio</code>	82
6.3	Miscellaneous information functions on <code>ratios</code>	82
6.4	Arithmetic operations for <code>ratios</code>	83
6.5	Rounding <code>ratios</code>	84
6.6	Comparisons between and with <code>ratios</code>	86
6.7	Coercion functions on <code>ratios</code>	87
6.7.1	Coercion between <code>ints</code> and <code>ratios</code>	87
6.7.2	Coercion between <code>nats</code> and <code>ratios</code>	87
6.7.3	Coercion between <code>big_ints</code> and <code>ratios</code>	88
6.7.4	Coercion between floating point numbers and <code>ratios</code>	88
6.7.5	Coercion between strings and <code>ratios</code>	88
6.8	Miscellaneous power functions	92
6.9	Floating point approximations of rational numbers	92
7	Rational numbers of type <code>num</code>	95
7.1	Normalization of <code>nums</code>	95
7.2	Miscellaneous information functions on <code>nums</code>	95
7.3	Arithmetic operations for <code>nums</code>	96
7.3.1	Incrementing and decrementing	96
7.3.2	Arithmetic operations for <code>nums</code>	96
7.4	Rounding <code>nums</code>	97

7.5	Comparisons between <code>nums</code>	98
7.6	Coercion functions on <code>nums</code>	99
7.6.1	Coercion between <code>ints</code> and <code>nums</code>	99
7.6.2	Coercion between floating point numbers and <code>nums</code>	100
7.6.3	Coercion between <code>nats</code> and <code>nums</code>	100
7.6.4	Coercion between <code>big_ints</code> and <code>nums</code>	101
7.6.5	Coercion between <code>ratios</code> and <code>nums</code>	101
7.6.6	Coercion between strings and <code>nums</code>	101
7.7	Miscellaneous power functions	106
7.8	Floating point approximations of <code>nums</code>	106
8	Numerical directives	109
8.1	Standard arithmetic	109
8.2	Fast arithmetic	109
8.3	Open arithmetic	110
8.4	Debugging nat arithmetic	110
8.5	Arithmetic overloading	110
8.6	Normalization of rational numbers during computation	111
8.7	Normalization of rational numbers when printing	111
8.8	Floating point approximation when printing rational numbers	111
8.9	Error when a rational denominator is null	112
8.10	<code>Arith_status</code> or how to remember (or use) all this	112
A	Numerical computations using CAML: π as a case-study	115
A.1	Introduction	115
A.2	First approach	115
A.2.1	Factorial function	115
A.2.2	Square root with a given precision	116
A.2.3	Required number of terms for a given precision	116
A.2.4	The program	118
A.2.5	Running the program	118
A.3	Second approach	119
A.3.1	The algorithm	119
A.3.2	Stop	120
A.3.3	The program	121
A.3.4	Running the program	122
A.4	Third approach	122
A.4.1	The algorithm	122
A.4.2	The program	123
A.4.3	Running the program	124
A.5	Fourth approach	124
A.5.1	The algorithm	124
A.5.2	The program	125
A.5.3	Running the program	125
A.5.4	<code>big_int</code> version	126

A.5.5	Running the program	128
A.6	Final version: type <code>nat</code>	128
A.6.1	Number of iterations	128
A.6.2	Size of the objects with base 2^{32}	130
A.6.3	The complete program	132
A.6.4	Running the program	135
B	To normalize or not to normalize rational numbers?	137
B.1	On average not to normalize is the better strategy	137
B.2	Exception	138
B.2.1	Exposition	138
B.2.2	Floating point version	138
B.2.3	The program	139

Chapitre 1

Introduction

1.1 Casual user

First of all, here are information for the casual user. It can be used in the three following modes (for more details see chapter 8).

In the standard arithmetic mode (which is the default one), numbers are supposed to be either integers in the range [$monster_int = -32768 \dots biggest_int = 2^{length_of_int} - 1 = 32767$] or floating point numbers. Standard arithmetic operations (+, -, *, /, ...) are defined on integers.

Example

```
#1+2;;  
3 : int
```

```
#1.0+2.0;;
```

```
line 1: ill-typed phrase, the constant 1.0 of type float  
cannot be used with type instance int in 1.0+2.0
```

```
line 1: ill-typed phrase, the constant 2.0 of type float  
cannot be used with type instance int in 1.0+2.0  
2 errors in typechecking
```

Typecheck Failed

If you want to use arithmetic operators for both integers and floating point numbers, you can overload these operators using the pragma:

Example

```
##arith overloading;;  
Symbol prefix + overloaded with  
  add_num : num * num -> num,  
  add_float : float * float -> float,  
  add_int : int * int -> int  
Symbol prefix * overloaded with  
  mult_num : num * num -> num,  
  mult_float : float * float -> float,  
  mult_int : int * int -> int
```

```
Symbol prefix ** overloaded with
  power_num : num * num -> num,
  power_float : float * float -> float,
  power_int : int * int -> int
Symbol prefix - overloaded with
  sub_num : num * num -> num,
  sub_float : float * float -> float,
  sub_int : int * int -> int
Symbol prefix / overloaded with
  div_num : num * num -> num,
  div_float : float * float -> float,
  div_int : int * int -> int
Symbol prefix <= overloaded with
  le_num : num * num -> bool,
  le_float : float * float -> bool,
  le_int : int * int -> bool
Symbol prefix < overloaded with
  lt_num : num * num -> bool,
  lt_float : float * float -> bool,
  lt_int : int * int -> bool
Symbol prefix >= overloaded with
  ge_num : num * num -> bool,
  ge_float : float * float -> bool,
  ge_int : int * int -> bool
Symbol prefix > overloaded with
  gt_num : num * num -> bool,
  gt_float : float * float -> bool,
  gt_int : int * int -> bool
Symbol - overloaded with
  minus_num : num -> num,
  minus_float : float -> float,
  minus_int : int -> int
Symbol float overloaded with
  float_of_num : num -> float,
  float_of_int : int -> float
Symbol pred overloaded with
  pred_num : num -> num,
  pred_float : float -> float,
  pred_int : int -> int
Symbol succ overloaded with
  succ_num : num -> num,
  succ_float : float -> float,
  succ_int : int -> int
Symbol incr overloaded with
  incr_num : num ref -> num,
  incr_float : float ref -> float,
  incr_int : int ref -> int
Symbol decr overloaded with
  decr_num : num ref -> num,
  decr_float : float ref -> float,
  decr_int : int ref -> int
```

```

Symbol prefix quo overloaded with
  quo_num : num * num -> num,
  quo_int : int * int -> int
Symbol prefix mod overloaded with
  mod_num : num * num -> num,
  mod_int : int * int -> int
Symbol max overloaded with
  max_num : num -> num -> num,
  max_float : float -> float -> float,
  max_int : int -> int -> int
Symbol min overloaded with
  min_num : num -> num -> num,
  min_float : float -> float -> float,
  min_int : int -> int -> int
Symbol abs overloaded with
  abs_num : num -> num,
  abs_float : float -> float,
  abs_int : int -> int
Symbol prefix power overloaded with
  power_num : num * num -> num,
  power_float : float * float -> float,
  power_int : int * int -> int

```

```

#1+2;;
3 : int

```

```

#1.0/3.0 + 2.0/3.0;;
0.9999999 : float

```

```

#sin 1;;

```

```

line 1: ill-typed phrase, the constant 1 of type int
cannot be used with type instance float in sin 1
1 error in typechecking

```

```

Typecheck Failed

```

```

#sin 1.0;;
0.841471 : float

```

CAML features an exact rational arithmetic package: the last arithmetic mode is thus extended arithmetic: numbers are supposed to be of type `num` and all available arithmetic operations always return exact results (up to 78903 decimal digits!).

Example

```

##standard arith false;;
Pragma false : bool

```

```

#1+2;;
#{3} : num

```

```
#1.0/3.0 + 2.0/3.0;;
#{1/1} : num
```

```
#sin 1.0;;
```

```
line 1: ill-typed phrase, the constant 1 of type num
cannot be used with type instance float in sin 1
1 error in typechecking
```

Typecheck Failed

About numerical directives, we can already mention the `arith_status ();;` instruction that prints the current status of each arithmetic flag and recall how to manage it.

1.2 Regular user

Numbers consist of:

type int: integers of type verb”int” with less than ($length_of_int = 15$) bits, in the range ($monster_int = -32768, biggest_int = 2^{length_of_int} - 1 = 32767$),

type float: floating point representations of reals,

type nat: integers of type verb”nat”, on which every operation deals with arbitrary large unsigned integers, works by side-effects, allocates no storage and returns generally unit or a carry while storing results in its first argument.

In fact such a number is represented as a string. Since a string cannot contain more than $biggest_int - 2$ characters and since each character allocates a byte, thus 8 bits, a number of type `nat` can have at most

$$\left\lfloor \frac{biggest_int - 2}{\left\lceil \frac{length_of_digit}{8} \right\rceil} \right\rfloor = 8191$$

digits, and is therefore itself less than or equal to

$$2^{length_of_digit \times \left\lfloor \frac{biggest_int - 2}{\left\lceil \frac{length_of_digit}{8} \right\rceil} \right\rfloor} - 1 = 2^{262112} - 1,$$

using classical notation for floor ($\lfloor \]$) and ceiling ($\lceil \]$) functions. This number has 78903 decimal digits, that is to say it would spread on about 25 pages if it were printed in this document.

type big_int: arbitrary large signed integers (with the same limitation as for “natural” numbers), for which every operation allocates storage for the result,

type ratio: arbitrary large rational numbers, represented by a pair of big integers and a boolean indicating if this rational number is known to be normalized or not.

infinity (1/0) and *undefined* (0/0) are included as regular rational values. On request, computation with these numbers does not lead to runtime errors.

type num: symbolic representations of small integers, arbitrary large integers and rational numbers, with optimization of the representation of a number generally speaking. Most of the time the casual user will manipulate **num** values without worrying about their representations. More sophisticated users will take advantage of the definition of the **num** type:

```
type num = Int of int | Big_int of big_int | Ratio of ratio;;
```

Some functions are only available in a special arithmetic mode (for more information on this mode see chapter 8 on numerical directives). These functions are pointed out by a † after their name in the function index and when they are documented.

The CAML arithmetic implementation uses the BigNum package developed jointly by INRIA and Digital PRL.

This package provides nat operations and is structured in two layers:

- the kernel which contains code for the time-critical low-level operations,
- a set of functions written in high level languages which call the kernel.

A lisp implementation of the kernel exists. However, to obtain a truly efficient implementation on a given machine, it must be written in assembly code. Unfortunately this assembly code implementation is not provided for all architectures.

The distinction between kernel and non-kernel operations is defined so that the time penalty for running the mixture of assembly and high level code, as opposed to pure assembly code, is less than 20% on typical benchmarks.

The kernel functions will be indicated in the corresponding chapter by a * character.

Chapitre 2

Operations on numbers of type `int`



```
length_of_int†: int  
biggest_int : int  
least_int   : int  
monster_int : int
```

Elements of type `int` are small integers represented on `length_of_int` bits, in the range $(\text{least_int}, \text{biggest_int})$, more exactly the ring of the integers between `least_int - 1` and `biggest_int`, since there is no notion of overflow.

This special value `monster_int = least_int - 1` will be further discussed (see section 2.2).

Integer values of type `int` are manipulated for the sake of efficiency (they are not allocated, hence there is no need to garbage collect them).

The maximum number of bits, the maximum, minimum and extremum values of an `int` are:

```
#length_of_int;;  
15 : int  
  
#biggest_int;;  
32767 : int  
  
#least_int;;  
-32767 : int  
  
#monster_int;;  
-32768 : int
```

2.1 Arithmetic operations for ints

2.1.1 Incrementing and decrementing

Incrementing and decrementing ints



```

pred_int : int → int
succ_int : int → int
monster_int : int

```

`pred_int n` (resp. `succ_int n`) is a primitive equivalent to `n-1` (resp. `n+1`).

`pred_int least_int` and `succ_int biggest_int` have the same value `monster_int`.

`pred_int monster_int` is `biggest_int` and `succ_int monster_int` is `least_int`.

The lexical convention for type `int` will be presented later in this chapter (section 2.3.3) but the essential thing to understand the following example is that elements of type `int` are preceded by a `#`.

Example

```

#pred_int #1;;
0 : int

#pred_int #0;;
-1 : int

#pred_int least_int;;
-32768 : int

#pred_int biggest_int;;
32766 : int

#pred_int monster_int;;
32767 : int

#succ_int #1;;
2 : int

#succ_int #0;;
1 : int

#succ_int least_int;;
-32766 : int

#succ_int biggest_int;;
-32768 : int

#succ_int monster_int;;
-32767 : int

```

Incrementing and decrementing counters



```

incr_int : int ref → int
decr_int : int ref → int

```

When one has created a reference containing a number (whose type is `int`), it is convenient to increment and decrement its value, using these two (primitive) functions semantically equivalent to:

```
let incr_int i = i := succ_int !i and decr_int i = i := pred_int !i;;
```

2.1.2 Arithmetic operations for ints



```
add_int : int × int → int
minus_int : int → int
sub_int : int × int → int
mult_int : int × int → int
quo_int : int × int → int
mod_int : int × int → int
gcd_int† : int × int → int
```

`add_int`, `sub_int`, `mult_int` perform respectively the addition, subtraction and multiplication on numbers of type `int`.

`minus_int x` is the opposite of `x`. We have :

$$\text{minus_int } \text{monster_int} = \text{monster_int}.$$

For division on numbers of type `int`, `quo_int (x, y)` and `mod_int (x, y)` are respectively the quotient and the remainder of the Euclidian division of `x` by `y`.

Beware:

→ In fact, in case of a negative argument, `quo_int (x, y)` is one more greater than the result of Euclidian division (i.e. each partial function of `quo_int` is an odd function) and `mod_int` doesn't always produce positive integers.

`gcd_int (n1, n2)` computes the gcd of the two numbers of type `int` `n1` and `n2`. We have the following relation:

$$\text{gcd_int}(n_1, n_2) = \text{gcd_int}(|n_1|, |n_2|) = \text{gcd_int}(n_2, n_1) \leq 0$$

Example

```
#add_int (#43, #-12);;
31 : int
```

```
#minus_int biggest_int;;
-32767 : int
```

```
#minus_int least_int;;
32767 : int
```

```
#minus_int monster_int;;
```

```

-32768 : int

#mult_int (biggest_int, #2);
-2 : int

#quo_int (#123, #3);
41 : int

#mod_int (#123, #3);
0 : int

#quo_int (#123, #-3);
-41 : int

#mod_int (#123, #-3);
0 : int

#quo_int (#123, #4);
30 : int

#mod_int (#123, #4);
3 : int

#quo_int (#123, #-4);
-30 : int

#mod_int (#123, #-4);
3 : int

#gcd_int (#4, #6);
2 : int

#gcd_int (#2, #3);
1 : int

#gcd_int (#-6, #4);
2 : int

```

Notice that these arithmetic operations do not check for overflow.

Using type `num`, a careful (but less efficient) version of multiplication may be defined as in:

```

#let mult_int_care (x, y) =
# match mult_num (Int x, Int y)
# with Int result -> result
# | _ -> failwith "overflow"
#;;
Value mult_int_care is <fun> : int * int -> int

#mult_int_care(biggest_int,#2);

Evaluation Failed: failure "overflow"

```

2.2 Comparisons between ints



```

eq_int : int × int → bool
compare_int†: int × int → int
lt_int : int × int → bool
le_int : int × int → bool
gt_int : int × int → bool
ge_int : int × int → bool
sign_int†: int → int
abs_int : int → int
minus_int : int → int
max_int : int → int → int
min_int : int → int → int

```

`eq_int` is the semantic and syntactic equality on numbers of type `int`.

`lt_int`, `le_int`, `gt_int`, `ge_int` are arithmetic comparisons corresponding respectively to `<`, `<=`, `>`, `>=`.

The primitive function for these functions is `compare_int`, that is semantically (and almost really) equivalent to

```

let compare_int (int1, int2) =
  when (int1 = int2) -> #0
    | (int1 < int2) -> #-1
    | _ -> #1;;

```

Example

```

#compare_int (#0, #1);;
-1 : int

```

```

#compare_int (#0, #0);;
0 : int

```

```

#compare_int (#0, #-1);;
1 : int

```

`sign_int` is completely equivalent to

```
let sign_int n = compare_int (n, 0);;
```

`abs_int n` is the absolute value of number `n`.

Beware:

→ `abs_int monster_int = minus_int monster_int = monster_int` and `monster_int` is negative. So `abs_int` and `minus_int` of a negative number of type `int` don't always produce positive numbers.

→ `pred_int monster_int = biggest_int` is the greatest number of type `int`, but `monster_int` is the least one. In fact `least_int` is the smallest “normal” number of type `int`.

`max_int n1 n2` and `min_int n1 n2` are respectively the greater and the smaller of the numbers `n1` and `n2`.

Example

```
#eq_int (succ_int biggest_int, monster_int);
true : bool
```

```
#gt_int (succ_int biggest_int, biggest_int);
false : bool
```

```
#sign_int (succ_int biggest_int);
-1 : int
```

```
#sign_int monster_int;
-1 : int
```

```
#sign_int #0;
0 : int
```

```
#sign_int #2;
1 : int
```

```
#sign_int #-3;
-1 : int
```

```
#abs_int #-3;
3 : int
```

```
#abs_int monster_int;
-32768 : int
```

```
#max_int #1 #2;
2 : int
```

```
#min_int #1 #2;
1 : int
```

2.3 Coercion functions on ints

2.3.1 Coercion between type float and int



`int_of_float : float → int`
`float_of_int : int → float`

`int_of_float f` is the truncation of `f` to a number of type `int`, closest to its floating point representation.

Example

```
#int_of_float #1.2;
1 : int
```

```
#int_of_float #1.0;;
1 : int

#int_of_float #0.9999999;;
0 : int

#int_of_float #0.99999999;;
1 : int
```

`float_of_int i` is the floating representation of the number of type `int i`.

Example

```
#float_of_int #0;;
0.0 : float

#float_of_int #1;;
1.0 : float

#float_of_int #-1;;
-1.0 : float
```

2.3.2 Coercion between characters and ints



```
int_of_char : char → int
char_of_int : int → char
```

`int_of_char c` is the ASCII code for `c`.

`char_of_int i` is the character with ASCII code `i`.

The two following relations hold for each `i` from type `int` and each `c` from type `char` :

$$\begin{aligned} \text{int_of_char} (\text{char_of_int } i) &= i \\ \text{char_of_int} (\text{int_of_char } c) &= c \end{aligned}$$

Example

```
#char_of_int #49;;
'1' : char

#int_of_char '1';;
49 : int
```

2.3.3 Coercion between strings and ints

Reading ints



sys_int_of_string[†]: $\text{int} \times \text{string} \times \text{int} \times \text{int} \rightarrow \text{int}$
int_of_string : $\text{string} \rightarrow \text{int}$

CAML is able to read **ints** entered directly, when you type in a number preceded by a sharp “#”, following the lexical convention:

```
INT ::= '#' {'-' | '+'} Digit+
```

Example

```
##1;;
1 : int

##-1;;
-1 : int

##+1;;
1 : int

##32767;;
32767 : int

##32768;;
32768.0 : float

##-32767;;
-32767 : int

##-32768;;
-32768.0 : float
```

Using standard arithmetic, the sharp character is optional (see chapter 8 for more information).

sys_int_of_string (*base*, *s*, *off*, *len*) maps the substring *s_{off,len}* in base *base* ($2 \leq \text{base} \leq 16$) onto a number of type **int**.

int_of_string *s* is semantically equivalent to:

```
let int_of_string s =
  sys_int_of_string (#10, s, #0, length_string s)
;;
```

Example

```
#sys_int_of_string (#10, "123456789012345678989", #11, #5);;
23456 : int

#int_of_string "12345";;
12345 : int
```

Printing ints



```

string_of_int : int → string
string_for_read_of_int : int → string
sys_string_of_int†: int × string × int × string → string
display_int : int → unit
print_int : int → unit
print_int_for_read : int → unit

```

`sys_string_of_int` (`b`, `before`, `i`, `after`) writes into the resulting string first the string `before`, then the representation of `i` in base `b`, and lastly the string `after`.

`string_of_int` is semantically equivalent to:

```
let string_of_int i = sys_string_of_int (#10, "", i, "");;
```

`string_for_read_of_int` maps elements of type `int` onto corresponding elements of type `string` as well as `string_of_int`, but the sharp character `#` is added before the number according to the lexical convention for numbers of type `int`.

This function is semantically equivalent to:

```
let string_for_read_of_int i = sys_string_of_int (#10, "#", i, "");;
```

`display_int` prints in a trivial way (i.e. without formatting its argument) numbers of type `int`.

`print_int` is the formatting function for numbers of type `int`.

`print_int_for_read` prints numbers of type `int` such that they are readable by the CAML system.

On the one hand, `display_int` and `print_int` print numbers of type `int` without printing sharp character `#`. On the other hand, `print_int_for_read` prints numbers of type `int` with a sharp character `#`.

For more details on the differences between `display`- and `print`- and `print*_for_read`-like functions see chapter “Formatting Functions” in the reference manual.

Example

```

#sys_string_of_int (#2, "begin ", #20, " end");;
"begin 10100 end" : string

#sys_string_of_int (#16, "hello ", #20, " world");;
"hello 14 world" : string

#string_of_int #12345;;
"12345" : string

#string_for_read_of_int #12345;;
"#12345" : string

#print_int #10;;
10() : unit

#print_int_for_read #10;;
#10() : unit

```


2.4 Logical operations



```
land_int : int × int → int
lor_int  : int × int → int
lxor_int : int × int → int
lshift_int : int × int → int
lnot_int : int → int
```

These are “logical” functions which operate on numbers *bitwise* without modification of the arguments. All these functions are expanded on the fly into one or a few lines of assembly code.

Their meaning is:

land_int: The *n*th bit of `land_int (n1,n2)` has value 1 if and only if the *n*th bit of `n1` and the one of `n2` have both value 1, else 0.

Example

```
#land_int (#2, #4);;
0 : int
```

```
#land_int (#3, #8);;
0 : int
```

For instance to find if a number is a power of 2, you may define

```
#let is_power2 n = (n = land_int (n, -n)) ;;
Value is_power2 is <fun> : int -> bool
```

```
#is_power2 #4;;
true : bool
```

```
#is_power2 #3;;
false : bool
```

lor_int: The *n*th bit of `lor_int (n1,n2)` has value 0 if and only if the *n*th bit of `n1` and the one of `n2` have both value 0, else 1.

lxor_int: The *n*th bit of `lxor_int (n1,n2)` has value 0 if and only if the *n*th bit of `n1` and the one of `n2` have different values.

lshift_int: `lshift_int (n1,n2)` has value `n1` shifted of `n2` bits. If `n2` is positive the shift is done to the left (towards the most significant bits, hence it is a multiplication by 2^{n2}). If `n2` is negative the shift is done to the right (towards the least significant bits, then it is a division by 2^{-n2}).

Using `lshift_int` you may extract a bit field from an integer representation, as in:

```
#let extract n i = lshift_int (land_int (n, lshift_int (#1,i)), -i);;
Value extract is <fun> : int -> int -> int
```

```
#map (extract #24) (interval #0 #15);;
[0; 0; 0; 1; 1; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0] : int list

#map (extract (#-24)) (interval #0 #15);;
[0; 0; 0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1] : int list
```

lnot_int: A bit of `lnot_int` (`n`) has value 0 iff the corresponding bit of `n` has value 1, else 1.

2.5 Miscellaneous functions on type int



```
num_bits_int†: int → int
num_decimal_digits_int†: int → int
power_int : int × int → int
power_int_positive_int†: int × int → big_int
```

`num_bits_int i` (resp. `num_decimal_digits_int i`) is the number of digits of the binary (resp. decimal) representation of `i`.

`power_int (i, n)` (resp. `power_int_positive_int (i, n)`) is the `int` (resp. `big_int`) representation of i^n , with `n` any positive or null `int`.

In fact, as long as the result is an integer `power_int` doesn't fail.

Example

```
#num_bits_int #20;;
5 : int

#num_decimal_digits_int #20;;
2 : int

#power_int (#2, length_of_int);;
-32768 : int

#power_int (#2, succ_int length_of_int);;
0 : int

#power_int_positive_int (#2, length_of_int);;
#(32768) : big_int

#power_int_positive_int (#2, succ_int length_of_int);;
#(65536) : big_int

#power_int (#-1, #-2);;
1 : int
```


Chapitre 3

Floating point numbers

Elements of type `float` are floating point representations of decimal numbers, in a range depending on the machine which runs the CAML system but generally included in $[-2^{128} + 1; 2^{128} - 1]$ and in 10-radix representation, absolute value of exponent is generally less than 38. There exists beyond these numbers a value *Infinity* and many incorrect values (e.g. *NaN* standing for “Not a number”). For example, for the computer on which this document is preprocessed the biggest floating point number is almost $2^{128} - 1 \approx 3.4028235677e38$. So examples with `3.403e38` imply *Infinity*. Reading floating point numbers greater than $2^{1024} - 1 \approx 1.797693134862315e308$, or lesser than 10^{-340} but not null, produce an error and the message depends on the computer.

3.1 Arithmetic operations for floating point numbers

3.1.1 Incrementing and decrementing

Incrementing and decrementing numbers



```
pred_float : float → float
succ_float : float → float
```

`pred_float n` (resp. `succ_float n`) is a primitive equivalent to `n-1` (resp `n+1`).

Example

```
#pred_float #1.0;;
0.0 : float

#pred_float #0.0;;
-1.0 : float

#succ_float #1.0;;
2.0 : float

#succ_float #0.0;;
1.0 : float
```

Incrementing and decrementing counters



incr_float : float ref \rightarrow float
decr_float : float ref \rightarrow float

When one has created a reference containing a number (whose type is `float`), it is convenient to increment and decrement its value, using these two (primitive) functions semantically equivalent to:

```
let incr_float i = i := succ_float !i and decr_float i = i := pred_float !i;;
```

3.1.2 Arithmetic operations for floating point numbers



add_float : float \times float \rightarrow float
minus_float : float \rightarrow float
sub_float : float \times float \rightarrow float
mult_float : float \times float \rightarrow float
div_float : float \times float \rightarrow float

`add_float`, `sub_float`, `mult_float`, `div_float` perform respectively the addition, subtraction, multiplication and division on floating point numbers.

`minus_float x` is the opposite of `x`.

3.2 Comparisons between floating point numbers



eq_float : float \times float \rightarrow bool
lt_float : float \times float \rightarrow bool
le_float : float \times float \rightarrow bool
gt_float : float \times float \rightarrow bool
ge_float : float \times float \rightarrow bool
abs_float : float \rightarrow float
max_float : float \rightarrow float \rightarrow float
min_float : float \rightarrow float \rightarrow float

`eq_float` is the semantic and syntactic equality on floating point numbers.

`lt_float`, `le_float`, `gt_float`, `ge_float` are arithmetic comparisons corresponding respectively to `<`, `<=`, `>`, `>=`.

`abs_float n` is the absolute value of the number `n`.

`max_float f1 f2` and `min_float f1 f2` are respectively the greater and the smaller of the numbers `f1` and `f2`.

Example

```
#eq_float (#3.402e38, #3.403e38);;  
false : bool
```

```
#eq_float (#3.403e38, #3.404e38);;  
true : bool
```

```
#lt_float (#3.402e38, #3.403e38);;
true : bool

#lt_float (#3.403e38, #3.404e38);;
false : bool

#gt_float (#3.403e38, #3.404e38);;
false : bool

#abs_float #-3.0;;
3.0 : float

#max_float #1.0 #2.0;;
2.0 : float

#min_float #1.0 #2.0;;
1.0 : float
```

3.3 Floating point transcendental operations



```
sin : float → float
cos : float → float
asin : float → float
acos : float → float
atan : float → float
exp : float → float
log : float → float
log10 : float → float
sqrt : float → float
power_float : float × float → float
```

`sin` and `cos` are the classical trigonometric functions. As an example one can create a function `tan` using the mathematical definition:

Example

```
#let tan x = div_float (sin x, cos x);;
Value tan is <fun> : float -> float

#tan 1.0;;
1.557408 : float

#atan it;;
0.9999999 : float

#let pi_over_2 = div_float (3.14159265,2.0);;
Value pi_over_2 is 1.570796 : float

>(* Computation with floating point numbers approximates *)
#tan pi_over_2;;
```

```
13245400.0 : float
```

```

>(* But sometimes consistently ! *)
#atan it;;
1.570796 : float

```

`asin`, `acos` and `atan` are the reciprocal functions of these trigonometric functions.

`log` and `exp` are the Neperian logarithm and the exponential function. `log10` is the decimal logarithm.

`sqrt` is the square root function and `power_float` (`x`, `y`) yields x^y .

3.4 Coercion functions on floating point numbers

3.4.1 Reading floating point numbers



```

sys_float_of_string : int × string × int × int → float
float_of_string : string → float

```

CAML is able to read floating point numbers entered directly, following the lexical convention:

```

FLOAT ::= '#' {'-' | '+'} Digit+ '.' Digit+ {'e' {'-' | '+'} Digit+

```

Example

```

##1.0;;
1.0 : float

##1.0e2;;
100.0 : float

##1.0e-1;;
0.09999999 : float

##1.2e+2;;
120.0 : float

##-1.2e+2;;
-120.0 : float

##+1.2e+2;;
120.0 : float

```

Using standard arithmetic, the sharp character is optional (see chapter 8 for more information).

`sys_float_of_string` (`base`, `s`, `off`, `len`) maps the substring $s_{off, len}$ in base `base` ($2 \leq \text{base} \leq 16$) onto a floating point number.

`float_of_string s` is semantically equivalent to:

```

let float_of_string s =
  sys_float_of_string (#10, s, #0, length_string s)
;;

```

Example

```
#sys_float_of_string (#10, "123e-42", #0, #6);;
0.0123 : float
```

```
#float_of_string "123e-4";;
0.0123 : float
```

Printing floating point numbers

```
string_of_float : float → string
string_for_read_of_float : float → string
display_float : float → unit
print_float : float → unit
print_float_for_read : float → unit
```

`string_of_float` and `string_for_read_of_float` map elements of type `float` onto corresponding elements of type `string`. `string_for_read_of_float` adds the sharp character `#` before the number according to the lexical convention for numbers of type `float`.

`display_float` prints in a trivial way (i.e. without formatting its argument) floating point numbers.

`print_float` is the formatting function for floating point numbers.

`print_float_for_read` prints floating point numbers such that it is readable by the CAML system.

On the one hand, `display_float` and `print_float` print floating point numbers without printing sharp character `#`. On the other hand, `print_float_for_read` prints floating point numbers with sharp character `#`.

For more details on the differences between `display-` and `print-` and `print*_for_read`-like functions see chapter “Formatting Functions” in the reference manual.

Example

```
#string_of_float #123.45;;
"123.45" : string
```

```
#string_for_read_of_float #123.45;;
"#123.45" : string
```

```
#print_float #1.0;;
1.0() : unit
```

```
#print_float_for_read #1.0;;
#1.0() : unit
```


Chapitre 4

Natural numbers

Elements of type `nat` are quasi arbitrarily large unsigned numbers. All operations on this type are performed with in place update.

4.1 Natural numbers representation

- Numbers of type `nat` are base B decompositions of natural numbers :

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



`length_of_digit`[†]: int

`set_length_of_digit`[†]: int → unit

In fact, $B = 2^{\text{length_of_digit}}$ and in the current implementation, we have by default:

$$\text{length_of_digit} = 32.$$

This value can be modified using the function `set_length_of_digit`.

- We note here the subnat of

$$\text{nat} = \sum_{i=0}^{i < n} \text{nat}_i B^i$$

beginning at the *off*-th digit with length *len* digits

$$\text{nat}_{\text{off}, \text{len}} = \sum_{i=0}^{i < \text{len}} \text{nat}_{\text{off}+i} B^i.$$

The following conditions must be verified :

$$\begin{aligned} \text{off} &\geq 0 \\ \text{len} &\geq 1 \\ \text{off} + \text{len} &\leq n. \end{aligned}$$

These conditions are supposed to be verified and are tested only in cautious arithmetic mode (see chapter 8) with the following instruction:

```
#arith cautious;;
```

If not respected it can cause a segment violation.

- In a straightforward manner, the i -th digit of

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

is N_i . The following condition must be :

$$0 \leq i < n.$$

Beware:

→ Notice that all functions on type `nat` are accessible only in open arithmetic mode. So they are all followed by a †. For more information on the open arithmetic mode see chapter 8.

4.2 Creation of natural numbers



```
create_nat†: int → nat
make_nat†: int → nat
copy_nat†: nat × int × int → nat
```

`create_nat len` and `make_nat len` create a new natural number of len digits ($len \geq 1$). `create_nat` is the primitive function, it is not supposed to initialize the created number, on the other hand, `make_nat` is supposed to initialize the created number to zero.

`make_nat` is equivalent to the following CAML definition:

```
let make_nat len =
  let res = create_nat (len) in
  set_to_zero_nat (res, #0, len);
res
;;
```

Beware:

→ In fact in this version `create_nat` and `make_nat` are identical and initialize the created number.

`copy_nat (nat, off, len)` is a copy of the value of subnat $nat_{off, len}$.

4.3 Assignment of natural numbers



set_digit_nat^{†*}: $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$

blit_nat^{†*}: $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$

set_to_zero_nat^{†*}: $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$

`set_digit_nat (nat, off, digit)` assigns nat_{off} to the value of digit of type `int`.

`blit_nat (nat1, off1, nat2, off2, len2)` assigns subnat $\text{nat2}_{\text{off2}, \text{len2}}$ to the subnat $\text{nat1}_{\text{off1}, \text{len2}}$.

The following conditions must be verified :

$$\begin{aligned} \text{off1} &\geq 0 \\ \text{off2} &\geq 0 \\ \text{len2} &> 0 \\ \text{off1} + \text{len2} &\leq \text{size}(\text{nat1}) \\ \text{off2} + \text{len2} &\leq \text{size}(\text{nat2}) \end{aligned}$$

where $\text{size}(\text{nat})$ designates the length of nat .

Notice that nat1 and nat2 can be the same.

`set_to_zero_nat (nat, off, len)` sets subnat $\text{nat}_{\text{off}, \text{len}}$ to zero.

Example

```
#let nat1 = create_nat (#2)
#and nat2 = create_nat (#2);;
Value nat1 is #<0> : nat
Value nat2 is #<0> : nat

#set_to_zero_nat (nat1, #0, #2);
#set_digit_nat (nat2, #0, #1);
#set_digit_nat (nat2, #1, #1);
#debug_print_nat nat1;;
|00000000|00000000|() : unit

#debug_print_nat nat2;;
|00000001|00000001|() : unit

#blit_nat (nat1, #0, nat2, #1, #1);
#debug_print_nat nat1;;
|00000000|00000001|() : unit

#debug_print_nat nat2;;
|00000001|00000001|() : unit

#blit_nat (nat1, #1, nat1, #0, #1);
#debug_print_nat nat1;;
|00000001|00000001|() : unit

#debug_print_nat nat2;;
|00000001|00000001|() : unit
```

4.4 Lengths of a natural number



`length_nat`[†]: `nat` → `int`
`num_digits_nat`^{†*}: `nat` × `int` × `int` → `int`

`length_nat nat` is the presupposed length of `nat` and `num_digits_nat (nat, off, len)` is the real length of `subnat natoff,len`.

Example

```
#let nat = create_nat (#2) in
# set_digit_nat (nat, #0, #1);
# set_digit_nat (nat, #1, #0);
# (length_nat nat, num_digits_nat (nat, #0, #2))
#;;
(2,1) : int * int
```

4.5 Miscellaneous informations on natural numbers



`is_digit_int`[†]: `nat` × `int` → `bool`
`is_digit_normalized`^{†*}: `nat` × `int` → `bool`
`is_digit_zero`^{†*}: `nat` × `int` → `bool`
`is_digit_odd`[†]: `nat` × `int` → `bool`
`is_zero_nat`[†]: `nat` × `int` × `int` → `bool`
`num_leading_zero_bits_in_digit`^{†*}: `nat` × `int` → `int`
`is_nat_int`[†]: `nat` × `int` × `int` → `bool`
`nth_digit_nat`^{†*}: `nat` × `int` → `int`

`is_digit_int (nat, off)` tests if `natoff` can be coerced to an `int`, i.e. this digit has less than `length_of_int` bits, i.e. is smaller than `biggest_int`.

`is_digit_normalized (nat, off)` returns true if

$$nat_{off} \geq \frac{B}{2}.$$

`is_digit_zero (nat, off)` returns true if `natoff = 0`.

`is_digit_odd (nat, off)` returns true if `natoff` is odd.

`is_zero_nat (nat, off, len)` tests if `subnat natoff,len` is null.

`num_leading_zero_bits_in_digit (nat, off)` returns the number of most significant bits consecutively null of `natoff`.

`is_nat_int (nat, off, len)` tests if `natoff,len` can be coerced in an `int`.

`nth_digit_nat (nat, off)` returns `natoff`.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done;;
```

```

() : unit

#debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#is_digit_int (nat, #0);;
true : bool

#is_nat_int (nat, #0, #2);;
false : bool

#nth_digit_nat (nat, #1);;
1 : int

#is_digit_normalized (nat, #1);;
false : bool

#is_digit_zero (nat, #0);;
true : bool

#is_digit_zero (nat, #1);;
false : bool

#is_digit_odd (nat, #1);;
true : bool

#is_digit_odd (nat, #2);;
false : bool

#is_zero_nat (nat, #0, #1);;
true : bool

#is_zero_nat (nat, #0, #4);;
false : bool

#num_leading_zero_bits_in_digit (nat, #0);;
32 : int

#num_leading_zero_bits_in_digit (nat, #1);;
31 : int

```

Notice that `num_leading_zero_bits_in_digit` can be used to define most of these preceding functions as follows:

```

let is_digit_normalized (nat, off) =
  num_leading_zero_bits_in_digit (nat, off) == 0
;;

let is_digit_zero (nat, off) =
  num_leading_zero_bits_in_digit (nat, off) == length_of_digit
;;

```

```

let is_zero_nat (nat, off, len) =
  num_digits_nat (nat, off, len) == 1 &&
  num_leading_zero_bits_in_digit (nat, off) >= length_of_digit
;;

let is_digit_int (nat, off) =
  num_leading_zero_bits_in_digit (nat, off) >= length_of_digit - length_of_int
;;

let is_nat_int (nat, off, len) =
  num_digits_nat (nat, off, len) == 1 &&
  num_leading_zero_bits_in_digit (nat, off) >= length_of_digit - length_of_int
;;

```

In practice these functions are not so defined and their existence gives less interest to `num_leading_zero_bits_in_digit` since all the significant information of this function is present more pleasantly in the other functions.

4.6 Arithmetic operations on natural numbers

4.6.1 Addition operators on natural numbers



`incr_nat`^{†*}: $\text{nat} \times \text{int} \times \text{int} \times \text{int} \rightarrow \text{int}$
`add_nat`^{†*}: $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \times \text{int} \rightarrow \text{int}$

Incrementing natural numbers

`incr_nat` (`nat`, `off`, `len`, `c_in`) adds in place the carry in `c_in` (`c_in` = 0 or `c_in` = 1) to the subnat `natoff,len` and returns the carry out `c_out`.

More formally speaking `incr_nat` performs the following operation :

$$\begin{aligned}
 \text{nat}_{\text{off},\text{len}} + c_{\text{in}} &= \\
 &\left(\sum_{i=0}^{i<\text{len}} \text{nat}_{\text{off}+i} B^i \right) + c_{\text{in}} = \\
 &\left(\sum_{i=0}^{i<\text{len}} \text{nat}'_{\text{off}+i} B^i \right) + c_{\text{out}} B^{\text{len}}.
 \end{aligned}$$

`incr_nat` replaces each digit `natoff+i` with its equivalent digit `nat'off+i` and returns the carry out `c_out`, with value 0 or 1.

`c_out` = 1 if and only if `natoff,len` = $B^{\text{len}} - 1$ and `c_in` = 1. `len` may be null, so the carry out equals the carry in.

Example

```

#let nat = make_nat #4;;
Value nat is #<0> : nat

```

```

#complement_nat (nat, #0, #4);
() : unit

#debug_print_nat nat;;
|FFFFFFFF|FFFFFFFF|FFFFFFFF|FFFFFFFF|() : unit

#incr_nat (nat, #2, #2, #1);
1 : int

#debug_print_nat nat;;
|00000000|00000000|FFFFFFFF|FFFFFFFF|() : unit

#incr_nat (nat, #0, #3, #1);
0 : int

#debug_print_nat nat;;
|00000000|00000001|00000000|00000000|() : unit

#incr_nat (nat, #0, #3, #1);
0 : int

#debug_print_nat nat;;
|00000000|00000001|00000000|00000001|() : unit

#incr_nat (nat, #0, #3, #1);
0 : int

#debug_print_nat nat;;
|00000000|00000001|00000000|00000002|() : unit

```

Addition on natural numbers

`add_nat (nat1, off1, len1, nat2, off2, len2, c_in)` adds subnats $nat1_{off1, len1}$, $nat2_{off2, len2}$ and the carry in c_{in} , places result in subnat $nat1_{off1, len1}$, propagates this carry on subnat $nat1_{off1+len2, len1-len2}$ and returns the carry out c_{out} .

More formally speaking `add_nat` performs the following operation :

$$\begin{aligned}
 & nat1_{off1, len1} + nat2_{off2, len2} + c_{in} = \\
 & \left(\sum_{i=0}^{i < len1} nat1_{off1+i} B^i \right) + \left(\sum_{i=0}^{i < len2} nat2_{off2+i} B^i \right) + c_{in} = \\
 & \left(\sum_{i=0}^{i < len1} nat'_{off1+i} B^i \right) + c_{out} B^{len1}.
 \end{aligned}$$

`add_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit nat'_{off1+i} and returns the carry out c_{out} , with value 0 or 1. This implies $0 \leq c_{in} \leq 1$ and $len1 \geq len2$.

$nat1$ and $nat2$ can be physically the same one under the condition $off1 \leq off2$.

If $len2 = 0$, `add_nat (nat1, off1, len1, nat2, off2, len2, c_in)` is equivalent to `incr_nat (nat1, off1, len1, c_in)`.

Example


```

#let nat1 = nat_of_int #2;;
Value nat1 is #<2> : nat

#add_nat (nat1, #0, #1, nat_of_int #3, #0, #1, #0);;
0 : int

#debug_print_nat nat1;;
|00000005|() : unit

#add_nat (nat1, #0, #1, nat_of_int #3, #0, #1, #1);;
0 : int

#debug_print_nat nat1;;
|00000009|() : unit

#let nat1 = nat_of_int #3;;
Value nat1 is #<3> : nat

#complement_nat (nat1, #0, #1);;
() : unit

#add_nat (nat1, #0, #1, nat_of_int #3, #0, #1, #1);;
1 : int

#debug_print_nat nat1;;
|00000000|() : unit

#let nat1 = create_nat #3;;
Value nat1 is #<0> : nat

#for i = #0 to #1 do set_digit_nat (nat1, i, i) done;
#set_digit_nat (nat1, #2, #0);
#complement_nat (nat1, #0, #2);
#debug_print_nat nat1;;
|00000000|FFFFFFFF|FFFFFFFF|() : unit

#add_nat (nat1, #0, #3, nat1, #1, #1, #0);;
0 : int

#debug_print_nat nat1;;
|00000000|FFFFFFFF|FFFFFFFD|() : unit

#let nat2 = copy_nat (nat1, #0, #3);;
Value nat2 is #<18446744073709551613> : nat

#add_nat (nat1, #0, #2, nat1, #1, #1, #0);;
1 : int

#debug_print_nat nat1;;
|00000000|00000000|FFFFFFFC|() : unit

```

```
#add_nat (nat2, #0, #3, nat2, #1, #1, #0);
0 : int

#debug_print_nat nat2;;
|00000001|00000000|FFFFFFFF|() : unit
```

4.6.2 Subtraction operators on natural numbers



```
complement_nat†*: nat × int × int → unit
decr_nat†*: nat × int × int × int → int
sub_nat†*: nat × int × int × nat × int × int × int → int
```

Complement of natural numbers

`complement_nat (nat, off, len)` replaces each digit of the subnat $nat_{off,len}$ with its B -complement, corresponding to logical inversion of each bit of each concerned digit.

More formally speaking `complement_nat` performs the following operation :

$$\overline{nat_{off,len}} = \sum_{i=0}^{i < len} \overline{nat_{off+i}} B^i = \sum_{i=0}^{i < len} \overline{nat_{off+i}} B^i = \sum_{i=0}^{i < len} (B - nat_{off+i} - 1) B^i.$$

No side effect is performed if $len = 0$.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do
# set_digit_nat (nat, i, i)
#done;
#debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#complement_nat (nat, #1, #2);
#debug_print_nat nat;;
|00000003|FFFFFFFD|FFFFFFFE|00000000|() : unit
```

Decrementing natural numbers

`decr_nat (nat, off, len, b_in)` subtracts in place the borrow in b_in ($b_in = 0$ or $b_in = 1$) to the subnat $nat_{off,len}$ and returns the borrow out b_out .

More formally speaking `decr_nat` performs the following operation :

$$nat_{off,len} + B^{len} + b_in - 1 = \left(\sum_{i=0}^{i < len} nat_{off+i} B^i \right) + B^{len} + b_in - 1 =$$

$$\left(\sum_{i=0}^{i < len} nat'_{off+i} B^i \right) + b_out B^{len}.$$

`decr_nat` replaces each digit nat_{off+i} with its equivalent digit nat'_{off+i} and returns the borrow out b_out , with value 0 or 1.

$b_out = 0$ if and only if $nat_{off, len} = 0$ and $b_in = 0$. len may be null, so the borrow out equals the borrow in.

Example

```
#let nat = make_nat #4;;
Value nat is #<0> : nat

#set_digit_nat (nat, #3, #1);;
() : unit

#debug_print_nat nat;;
|00000001|00000000|00000000|00000000|() : unit

#decr_nat (nat, #0, #4, #1);;
1 : int

#debug_print_nat nat;;
|00000001|00000000|00000000|00000000|() : unit

#decr_nat (nat, #0, #4, #0);;
1 : int

#debug_print_nat nat;;
|00000000|FFFFFFFF|FFFFFFFF|FFFFFFFF|() : unit

#decr_nat (nat, #0, #4, #0);;
1 : int

#debug_print_nat nat;;
|00000000|FFFFFFFF|FFFFFFFF|FFFFFFFE|() : unit

#decr_nat (nat, #0, #4, #1);;
1 : int

#debug_print_nat nat;;
|00000000|FFFFFFFF|FFFFFFFF|FFFFFFFE|() : unit
```

Subtraction on natural numbers

`sub_nat` ($nat1$, $off1$, $len1$, $nat2$, $off2$, $len2$, b_in) subtracts $nat2_{off2, len2}$ and the borrow in b_in to $nat1_{off1, len1}$, places result in subnat $nat1_{off1, len1}$, propagates the borrow of this subtraction on subnat $nat1_{off1+len2, len1-len2}$ and returns the borrow out b_out .

More formally speaking `sub_nat` performs the following operation :

$$nat1_{off1, len1} - nat2_{off2, len2} + B^{len1} + b_in - 1 =$$

$$\left(\sum_{i=0}^{i < len1} nat1_{off1+i} B^i \right) - \left(\sum_{i=0}^{i < len2} nat2_{off2+i} B^i \right) + B^{len1} + b_in - 1 = \left(\sum_{i=0}^{i < len1} nat'_{off1+i} B^i \right) + b_out B^{len1}.$$

`sub_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit nat'_{off1+i} and returns the borrow out b_out , with value 0 or 1. This implies $0 \leq b_in \leq 1$ and $len1 \geq len2$.

$nat1$ and $nat2$ can be physically the same one under the condition $off1 \leq off2$.

If $len2 = 0$, `sub_nat` ($nat1$, $off1$, $len1$, $nat2$, $off2$, $len2$, b_in) is equivalent to `decr_nat` ($nat1$, $off1$, $len1$, b_in).

Example

```
#let nat = make_nat #4;;
Value nat is #<0> : nat

#complement_nat (nat, #3, #1);;
() : unit

#debug_print_nat nat;;
|FFFFFFFF|00000000|00000000|00000000|() : unit

#sub_nat (nat, #0, #1, nat, #1, #1, #0);;
0 : int

#debug_print_nat nat;;
|FFFFFFFF|00000000|00000000|FFFFFFFF|() : unit

#sub_nat (nat, #0, #2, nat, #2, #1, #0);;
1 : int

#debug_print_nat nat;;
|FFFFFFFF|00000000|00000000|FFFFFFFE|() : unit

#sub_nat (nat, #2, #1, nat, #3, #1, #0);;
0 : int

#debug_print_nat nat;;
|FFFFFFFF|00000000|00000000|FFFFFFFE|() : unit

#sub_nat (nat, #0, #3, nat, #0, #1, #0);;
0 : int

#debug_print_nat nat;;
|FFFFFFFF|FFFFFFFF|FFFFFFFF|FFFFFFFF|() : unit
```

4.6.3 Multiplication operators on natural numbers



mult_digit_nat : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{int}$
mult_nat : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$
shift_left_nat : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$
square_nat : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$

Multiplication of a natural number by a digit

`mult_digit_nat (nat1, off1, len1, nat2, off2, len2, nat3, off3)` multiplies `subnat nat2off2,len2` by digit `nat3off3`, adds the result to `nat1off1,len1` and returns the carry out `c_out`.

More formally speaking `mult_digit_nat` performs the following operation :

$$\begin{aligned}
 & \text{nat1}_{\text{off1},\text{len1}} + \text{nat3}_{\text{off3}} \times \text{nat2}_{\text{off2},\text{len2}} = \\
 & \left(\sum_{i=0}^{i<\text{len1}} \text{nat1}_{\text{off1}+i} B^i \right) + \text{nat3}_{\text{off3}} \left(\sum_{i=0}^{i<\text{len2}} \text{nat2}_{\text{off2}+i} B^i \right) = \\
 & \left(\sum_{i=0}^{i<\text{len1}} \text{nat}'_{\text{off1}+i} B^i \right) + c_out B^{\text{len1}}.
 \end{aligned}$$

`mult_digit_nat` replaces each digit `nat1off1+i` with its equivalent digit `nat'off1+i` and returns the carry out `c_out`, with value 0 or 1. This implies `len1 > len2`.

If `len2 = 0` and `c_out = 0` and no side effect is performed. `nat1` and `nat2` can be physically the same one under the condition `off1 ≤ off2`. The digit `nat3off3` may be any digit of `nat1` or `nat2`.

Beware:

→ For speed, the cases `nat3off3 = 0` and `nat3off3 = 1` are explicitly tested.

Example

```

#let nat1 = make_nat #5
#and nat2 = create_nat #4;;
Value nat1 is #<0> : nat
Value nat2 is #<0> : nat

#set_digit_nat (nat1, #1, #2);;
() : unit

#for i = #0 to #3 do set_digit_nat (nat2, i, i) done;;
() : unit

#debug_print_nat nat1;;
|00000000|00000000|00000000|00000002|00000000|() : unit

#debug_print_nat nat2;;
|00000003|00000002|00000001|00000000|() : unit

#mult_digit_nat (nat1, #0, #5, nat2, #0, #4, nat_of_int #15, #0);;
0 : int

```

```
#debug_print_nat nat1;;
|00000000|0000002D|0000001E|00000011|00000000|() : unit

#mult_digit_nat (nat1, #0, #5, nat2, #0, #4, nat2, #2);;
0 : int

#debug_print_nat nat1;;
|00000000|00000033|00000022|00000013|00000000|() : unit
```

Multiplication on natural numbers

`mult_nat (nat1, off1, len1, nat2, off2, len2, nat3, off3, len3)` multiplies subnats $nat2_{off2,len2}$ and $nat3_{off3,len3}$; adds the result to $nat1_{off1,len1}$ and returns the carry out c_out .

More formally speaking `mult_nat` performs the following operation :

$$\begin{aligned}
 nat1_{off1,len1} + nat2_{off2,len2} \times nat3_{off3,len3} = \\
 \left(\sum_{i=0}^{i < len1} nat1_{off1+i} B^i \right) + \left(\sum_{i=0}^{i < len2} nat2_{off2+i} B^i \right) \left(\sum_{i=0}^{i < len3} nat3_{off3+i} B^i \right) = \\
 \left(\sum_{i=0}^{i < len1} nat'_{off1+i} B^i \right) + c_out B^{len1}.
 \end{aligned}$$

`mult_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit nat'_{off1+i} and returns the carry out c_out , with value 0 or 1. This implies $len1 \geq len2 + len3$. If $len2 = 0$ and $c_out = 0$ no side effect is performed.

Beware:

→ $nat2$ and $nat3$ have quite similar roles, but for speed reasons it is preferable that $len2 \geq len3$,

→ No overlap is possible between $nat1_{off1,len1}$ and $nat2_{off2,len2}$ or $nat3_{off3,len3}$.

Example

```
#let nat1 = create_nat #4
#and nat2 = create_nat #4;;
Value nat1 is #<0> : nat
Value nat2 is #<0> : nat

#for i = #0 to #3 do
# set_digit_nat (nat2, i, i);
# set_digit_nat (nat1, i, add_int (i, #4))
#done;
#debug_print_nat nat2;;
|00000003|00000002|00000001|00000000|() : unit

#debug_print_nat nat1;;
|00000007|00000006|00000005|00000004|() : unit

#mult_nat (nat1, #0, #4, nat2, #0, #2, nat2, #2, #2);;
0 : int
```

```

#debug_print_nat nat1;;
|00000007|00000009|00000007|00000004|() : unit

#complement_nat (nat2, #0, #4);;
() : unit

#debug_print_nat nat2;;
|FFFFFFFC|FFFFFFFD|FFFFFFFE|FFFFFFF|() : unit

#mult_nat (nat1, #0, #4, nat2, #0, #2, nat2, #2, #2);;
1 : int

#debug_print_nat nat1;;
|00000003|00000008|0000000D|00000007|() : unit

```

Shifting left natural numbers

`shift_left_nat (nat1, off1, len1, nat2, off2, shift)` shifts left any digit of subnat $nat1_{off1, len1}$. The *shift* bits out of any digit replace the *shift* free bits of the next digit. *shift* zeros replace the *shift* free bits of the first digit and the least significant bits of digit $nat2_{off2}$ get back the *shift* bits out of the last digit. So `shift_left_nat` multiplies $nat1_{off1, len1}$ by 2^{shift} .

More formally speaking `shift_left_nat` performs the following operation :

$$\begin{aligned}
2^{shift} nat1_{off1, len1} = \\
2^{shift} \left(\sum_{i=0}^{i < len1} nat1_{off1+i} B^i \right) = \\
\left(\sum_{i=0}^{i < len1} nat1'_{off1+i} B^i \right) + nat2'_{off2} B^{len1}.
\end{aligned}$$

`shift_left_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit $nat1'_{off1+i}$ and $nat2_{off2}$ with $nat2'_{off2}$. This implies $0 \leq shift < length_of_digit$.

Beware:

→ For speed, the case $shift = 0$ is explicitly tested.

Example

```

#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done;;
() : unit

#debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#shift_left_nat (nat, #1, #3, nat, #0, #8);;
() : unit

```

```
#debug_print_nat nat;;
|00000300|00000200|00000100|00000000|() : unit

#shift_left_nat (nat, #1, #3, nat, #0, #16);;
() : unit

#debug_print_nat nat;;
|03000000|02000000|01000000|00000000|() : unit

#shift_left_nat (nat, #1, #3, nat, #0, #16);;
() : unit

#debug_print_nat nat;;
|00000200|00000100|00000000|00000300|() : unit
```

Squaring natural numbers

`square_nat (nat1, off1, len1, nat2, off2, len2)` squared subnat $nat2_{off2, len2}$, adds the result to $2 \text{ nat1}_{off1, len1}$ and returns the carry out `c_out`.

More formally speaking `square_nat` performs the following operation :

$$2 \text{ nat1}_{off1, len1} + \left(\text{nat2}_{off2, len2}^2 \right) =$$

$$\left(\sum_{i=0}^{i < len1} \text{nat1}_{off1+i} B^i \right) + \left(\sum_{i=0}^{i < len2} \text{nat2}_{off2+i} B^i \right)^2 =$$

$$\left(\sum_{i=0}^{i < len1} \text{nat}'_{off1+i} B^i \right) + c_out B^{len1}.$$

`square_nat` replaces each digit nat1_{off1+i} with its equivalent digit nat'_{off1+i} and returns the carry out `c_out`, with value 0 or 1. This implies $len1 \geq 2 \times len2$. If `len2` is null, `c_out` = 0 and no side effect is performed. No overlap is possible between $\text{nat1}_{off1, len1}$ and $\text{nat2}_{off2, len2}$.

Example

```
#let nat2 = create_nat #2
#and nat1 = make_nat #4;;
Value nat2 is #<0> : nat
Value nat1 is #<0> : nat

#set_digit_nat (nat2, #0, #1);
#set_digit_nat (nat2, #1, #2);
#debug_print_nat nat2;;
|00000002|00000001|() : unit

#square_nat (nat1, #0, #4, nat2, #0, #2);;
0 : int

#debug_print_nat nat1;;
|00000000|00000004|00000004|00000001|() : unit
```


4.6.4 Division operators on natural numbers



div_digit_nat^{†*}: $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{int}$

div_nat[†]: $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$

shift_right_nat^{†*}: $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$

gcd_int_nat[†]: $\text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$

gcd_nat[†]: $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$

Division of a natural number by a digit

`div_digit_nat (quo, offq, rem, offr, nat1, off1, len1, nat2, off2)` divides subnat $\text{nat1}_{\text{off1}, \text{len1}}$ by digit $\text{nat2}_{\text{off2}}$, places the quotient in subnat $\text{quo}_{\text{offq}, \text{len1}-1}$ and the remainder in digit rem_{offr} .

More formally speaking `div_digit_nat` satisfies the following equation :

$$\sum_{i=0}^{i < \text{len1}} \text{nat1}_{\text{off1}+i} B^i = \left(\sum_{i=0}^{i < \text{len1}-1} \text{quo}_{\text{offq}+i} B^i \right) \times \text{nat2}_{\text{off2}} + \text{rem}_{\text{offr}}$$

with $0 \leq \text{rem}_{\text{offr}} < \text{nat2}_{\text{off2}}$.

`div_digit_nat` implies $\text{len1} > 1$, $\text{nat1}_{\text{off1}+\text{len1}-1} < \text{nat2}_{\text{off2}}$ and $\text{length_nat}(\text{quo}) \geq \text{off1} + \text{len1} - 1$.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#div_digit_nat (nat, #0, nat, #2, nat, #0, #3, nat, #3);;
3 : int

#debug_print_nat nat;;
|00000003|00000000|AAAAAAB|00000000|() : unit

#div_digit_nat (nat, #0, nat, #2, nat, #0, #3, nat, #3);;
3 : int

#debug_print_nat nat;;
|00000003|00000000|38E38E39|00000000|() : unit

#div_digit_nat (nat, #0, nat, #2, nat, #0, #3, nat, #3);;
3 : int

#debug_print_nat nat;;
|00000003|00000002|12F684BD|AAAAAAA|() : unit
```

```
#div_digit_nat (nat, #0, nat, #2, nat, #1, #2, nat, #0);;
0 : int

#debug_print_nat nat;;
|00000003|12F684BF|12F684BD|00000003|() : unit
```

Division of natural numbers

`div_nat (nat1, off1, len1, nat2, off2, len2)` divides subnat $nat1_{off1, len1}$ by subnat $nat2_{off2, len2}$, places the quotient in the subnat $nat1_{off1+len2, len1-len2}$ and the remainder in subnat $nat1_{off1, len2}$.

More formally speaking `div_nat` verifies the following equation :

$$\sum_{i=0}^{i < len1} nat1_{off1+i} B^i = \left(\sum_{i=0}^{i < len1-len2} nat'_{off1+len2+i} B^i \right) \times \left(\sum_{i=0}^{i < len2} nat2_{off2+i} B^i \right) + \left(\sum_{i=0}^{i < len2} nat'_{off+i} B^i \right)$$

with

$$0 \leq \sum_{i=0}^{i < len2} nat'_{off1+i} B^i < \sum_{i=0}^{i < len2} nat2_{off2+i} B^i.$$

`div_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit nat'_{off1+i} . This implies $len1 \geq len2$ and $nat1_{off1+len1-1} < nat2_{off2+len2-1}$.

Example

```
#let nat = create_nat #5;;
Value nat is #<0> : nat

#for i = #0 to #4 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000004|00000003|00000002|00000001|00000000|() : unit

#div_nat (nat, #0, #3, nat, #3, #2);;
() : unit

#debug_print_nat nat;;
|00000004|00000003|7FFFFFFF|00000003|80000003|() : unit
```

Shifting right natural numbers

`shift_right_nat (nat1, off1, len1, nat2, off2, shift)` shifts right each digit of $nat1_{off1, len1}$. The *shift* bits out of any digit replace the *shift* free bits of the previous digit. *shift* zeros replace the *shift* free bits of the last digit and the most significant bits of digit $nat2_{off2}$ get back the *shift* bits out of the first digit. So `shift_right_nat` divides $nat1_{off1, len1}$ by 2^{shift} .

More formally speaking `shift_right_nat` verifies the following equation :

$$\sum_{i=0}^{i < len1} nat1_{off1+i} B^i = 2^{shift} \left(nat2'_{off2} B^{-1} + \sum_{i=0}^{i < len1} nat1'_{off1+i} B^i \right).$$

`shift_right_nat` replaces each digit $nat1_{off1+i}$ with its equivalent digit $nat1'_{off1+i}$ and $nat2_{off2}$ with $nat2'_{off2}$. This implies $0 \leq shift < length_of_digit$. For speed, the case $shift = 0$ is explicitly tested.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#shift_right_nat (nat, #1, #3, nat, #0, #8);;
() : unit

#debug_print_nat nat;;
|00000000|03000000|02000000|01000000|() : unit

#shift_right_nat (nat, #1, #3, nat, #0, #16);;
() : unit

#debug_print_nat nat;;
|00000000|00000300|00000200|00000000|() : unit

#shift_right_nat (nat, #1, #3, nat, #0, #16);;
() : unit

#debug_print_nat nat;;
|00000000|00000000|03000000|02000000|() : unit
```

Greatest common divisor between an int and a nat

`gcd_int_nat (i, nat, off, len)` computes the gcd of “small” int i and subnat $nat_{off,len}$ and puts it in digit nat_{off} . If $i = 0$ then the resulting value of the function is 1 and digit nat_{off} is unchanged, else the resulting value of the function is 0 and the gcd is placed in digit nat_{off} .

Example

```
#let nat = create_nat #2;;
Value nat is #<0> : nat

#set_digit_nat (nat, #0, #3);
#set_digit_nat (nat, #1, #2);
#debug_print_nat nat;;
|00000002|00000003|() : unit

#gcd_int_nat (#10, nat, #0, #2);;
0 : int

#debug_print_nat nat;;
|00000002|00000005|() : unit
```

Greatest common divisor between natural numbers

`gcd_nat (nat1, off1, len1, nat2, off2, len2)` places the gcd of subnats $nat1_{off1, len1}$ and $nat2_{off2, len2}$ in $nat1_{off1, len}$ where len is the resulting value of `gcd_nat`. $len2$ can be smaller than $len1$ unless subnat $nat1_{off1, len1}$ is null.

Example

```
#let nat = create_nat #3;;
Value nat is #<0> : nat

#for i = #0 to #2 do set_digit_nat (nat, i, i) done;
#debug_print_nat nat;;
|00000002|00000001|00000000|() : unit

#let len = gcd_nat (nat, #0, #3, nat, #0, #2);;
Value len is 2 : int

#debug_print_nat (copy_nat (nat, #0, len));;
|00000001|00000000|() : unit
```

4.7 Comparisons on natural numbers

```
compare_digits_nat : nat × int × nat × int → int
compare_nat†: nat × int × int × nat × int × int → int
eq_nat†: nat × int × int × nat × int × int → bool
le_nat†: nat × int × int × nat × int × int → bool
ge_nat†: nat × int × int × nat × int × int → bool
lt_nat†: nat × int × int × nat × int × int → bool
gt_nat†: nat × int × int × nat × int × int → bool
```

Comparison between digits

`compare_digits_nat (nat1, off1, nat2, off2)` returns -1 if digit $nat1_{off1}$ is less than digit $nat2_{off2}$, 0 if they are equal and 1 otherwise.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#compare_digits_nat (nat, #0, nat, #0);;
0 : int

#compare_digits_nat (nat, #0, nat, #1);;
-1 : int
```

```

#compare_digits_nat (nat, #1, nat, #0);;
1 : int

#complement_nat (nat, #0, #2);;
() : unit

#debug_print_nat nat;;
|00000003|00000002|FFFFFFFFE|FFFFFFFF|() : unit

#compare_digits_nat (nat, #0, nat, #0);;
0 : int

#compare_digits_nat (nat, #0, nat, #1);;
1 : int

#compare_digits_nat (nat, #1, nat, #0);;
-1 : int

```

Comparison between natural numbers

`compare_nat (nat1, off1, len1, nat2, off2, len2)` returns -1 if subnat $nat1_{off1, len1}$ is less than subnat $nat2_{off2, len2}$, 0 if they are equal and 1 otherwise.

Beware:

→ For speed, if $len1 \neq len2$ then $compare_nat(nat1, off1, len1, nat2, off2, len2) = compare_int(len1, len2)$. So if one of the indicated length is greater than the real length of the corresponding subnat, the result of this comparison may be wrong.

Example

```

#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #2 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000000|00000002|00000001|00000000|() : unit

#compare_nat (nat, #1, #2, nat, #0, #2);;
1 : int

#compare_nat (nat, #2, #2, nat, #2, #1);;
1 : int

#compare_nat (nat, #3, #2, nat, #3, #1);;
1 : int

```

`eq_nat`, `le_nat`, `ge_nat`, `lt_nat` and `gt_nat` can be defined as follows :

```

let eq_nat args = eq_int (compare_nat args, #0)
and le_nat args = le_int (compare_nat args, #0)
and ge_nat args = ge_int (compare_nat args, #0)
and lt_nat args = lt_int (compare_nat args, #0)
and gt_nat args = gt_int (compare_nat args, #0)
;;

```

4.8 Coercion functions on natural numbers

4.8.1 Coercion between types `int` and `nat`



`sys_int_of_nat` : `nat × int × int → int`

`int_of_nat`[†] : `nat → int`

`nat_of_int` : `int → nat`

`sys_int_of_nat (nat, off, len)` converts subnat $nat_{off, len}$ into an `int` when it is possible. `int_of_nat` converts its argument into an `int` when it is possible. This function is semantically equivalent to:

```
let int_of_nat nat = sys_int_of_nat (nat, #0, length_nat nat);;
```

Example

```
#let nat = create_nat #2;;
Value nat is #<0> : nat
```

```
#set_digit_nat (nat, #0, #1024);;
() : unit
```

```
#complement_nat (nat, #1, #1);;
() : unit
```

```
#debug_print_nat nat;;
|FFFFFFFF|00000400|() : unit
```

```
#sys_int_of_nat (nat, #0, #1);;
1024 : int
```

```
#sys_int_of_nat (nat, #1, #1);;
```

```
Evaluation Failed: failure "sys_int_of_nat"
```

```
#sys_int_of_nat (nat, #0, #2);;
```

```
Evaluation Failed: failure "sys_int_of_nat"
```

```
#int_of_nat (make_nat #2);;
0 : int
```

`nat_of_int` converts its argument into a `nat` unless it is negative.

Example

```
#nat_of_int #0;;
#<0> : nat
```

```
#nat_of_int #1;;
#<1> : nat
```

```
#nat_of_int #-1;;
```

Evaluation Failed: failure "nat_of_int"

4.8.2 Coercion between types float and nat



`sys_float_of_nat`[†]: $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{float}$
`float_of_nat` : $\text{nat} \rightarrow \text{float}$
`nat_of_float` : $\text{float} \rightarrow \text{nat}$

`sys_float_of_nat (nat, off, len)` converts subnat $\text{nat}_{off,len}$ into a floating point number. `float_of_nat` converts its argument into a floating point number. This function is equivalent to:

```
let float_of_nat nat = sys_float_of_nat (nat, #0, length_nat nat);;
```

`nat_of_float` converts its floating point argument into a natural number when it is a positive integer.

Example

```
#let nat = create_nat #2;;
Value nat is #<0> : nat

#set_digit_nat (nat, #1, #1);;
() : unit

#sys_float_of_nat (nat, #0, #1);;
0.0 : float

#sys_float_of_nat (nat, #1, #1);;
1.0 : float

#float_of_nat nat;;
4294967000.0 : float

#nat_of_float #1.0;;
#<1> : nat

#nat_of_float #1.0e10;;
#<9999999000> : nat

#nat_of_float #1.5;;
```

Evaluation Failed: failure "sys_big_int_of_string"

4.8.3 Coercion between types string and nat

Reading natural numbers



```

sys_nat_of_string : int × string × int × int → nat
nat_of_string : string → nat

```

It is not possible to read directly natural numbers, but coercion from strings exists.

`sys_nat_of_string (base, s, off, len)` maps the substring $s_{off, len}$ in base `base` onto a natural number.

A string must verify the following convention for recognition as a natural number:

```

Separator ::= ' ' | '\f' | '\n' | '\r' | '\t' | '\\'
Pseudodigit ::= Digit | Separator
NAT ::= Separator* Digit Pseudodigit*

```

`nat_of_string s` is equivalent to:

```

let nat_of_string s =
  sys_nat_of_string (#10, s, #0, length_string s)
;;

```

Example

```

#sys_nat_of_string (#16, "FF", #0, #2);;
#<255> : nat

```

```

#nat_of_string "123";;
#<123> : nat

```

Printing natural numbers



```

base_digit_of_char†: char × int → int
sys_string_of_digit†: nat × int → string
string_of_digit†: nat → string
sys_string_list_of_nat†: int × nat × int × int → string list
sys_string_of_nat : int × string × nat × int × int × string → string
string_of_nat†: nat → string
string_for_read_of_nat†: nat → string
sys_print_nat†: int × string × nat × int × int × string → unit
print_nat†: nat → unit
print_nat_for_read†: nat → unit
debug_string_vect_nat†: nat → string vect
debug_string_nat†: nat → string
debug_print_nat†: nat → unit

```

`base_digit_of_char (c, base)` verifies `c` is a correct `base`-digit (as regards to usual conventions only uppercase letters are authorized in case of `base` greater than 10) and give its decimal value.

`sys_string_of_digit (nat, off)` map digit nat_{off} onto the corresponding string in 10-radix representation.

`string_of_digit nat` map first digit nat_0 onto the corresponding string in 10-radix representation.

`sys_string_list_of_nat (base, nat, off, len)` maps base-radix representation of subnat $nat_{off,len}$ onto a list of strings, the first string corresponds to the beginning of the number, and so on. Usually one string is enough for representing nats but for example the longest `nat` needs 32 strings for printing its 2-radix representation.

`sys_string_list_of_nat` is the most general printing function. It takes more time than the other printing functions but it can print any natural number even when all other printing functions fail. So the CAML system when printing values of type `nat`, `big_int`, `ratio` or `num` uses this function.

`sys_string_of_nat (base, before, nat, off, len, after)` writes into the resulting string first the string `before`, then the representation of subnat $nat_{off,len}$ in base `base`, and last the string `after`. This function is optimized in base 10.

`string_of_nat nat` is semantically (but not practically for efficiency reasons) equivalent to:

```
let string_of_nat nat =
  sys_string_of_nat (#10, "", nat, #0, length_nat nat, "")
;;
```

`string_for_read_of_nat nat` is analogous to `string_of_nat nat` except that the number is written between `#<` and `>`, according to conventions to write natural numbers.

This function is equivalent to:

```
let string_for_read_of_nat nat =
  sys_string_of_nat (#10, "#<", nat, #0, length_nat nat, ">")
;;
```

Beware:

→ This convention is not properly speaking a lexical convention since no natural number can be read directly, even though between `#<` and `>`.

`sys_print_of_nat (base, before, nat, off, len, after)` print the successive strings of `sys_string_list_of_nat (base, nat, off, len)`, between string `before` and `after`.

`print_nat` and `print_nat_for_read nat` are equivalent to:

```
let print_nat nat =
  sys_print_nat (#10, "", nat, #0,
                num_digits_nat (nat, #0, length_nat nat), "")
;;

let print_nat_for_read nat =
  sys_print_nat (#10, "#<", nat, #0,
                num_digits_nat (nat, #0, length_nat nat), ">")
;;
```

`debug_string_vect_nat nat` and `debug_string_nat` map nat onto its 16-radix representation, without normalizing the number according to its real length. If the result cannot fit in only one string, `debug_string_nat nat` fails but `debug_string_vect_nat nat` is a vector of more than one string, the first string correspond to the end of the number, and so on.

`debug_print_nat nat` prints the successive strings of `debug_string_vect_nat nat`.

Example

```

#base_digit_of_char ('9', #10);;
9 : int

#base_digit_of_char ('A', #10);;

Evaluation Failed: failure "base_digit_of_char"

#base_digit_of_char ('A', #16);;
10 : int

#let nat = create_nat #2;;
Value nat is #<0> : nat

#set_digit_nat (nat, #0, #1);;
() : unit

#set_digit_nat (nat, #1, #2);;
() : unit

#string_of_digit nat;;
"1" : string

#sys_string_of_digit (nat, #1);;
"2" : string

>(* A good example for this function would have a really long result.
# The resulting string should have more than biggest_int digits so
# takes more than six pages, too long for doing it here but it works well *)
#sys_string_list_of_nat (#10, nat, #0, #1);;
#sys_string_of_nat (#16, "nat 16-radix representation = ", nat, #0, #2, "");;
["1"] : string list

#sys_string_of_nat (#2, "nat 2-radix representation = ", nat, #0, #2, "");;
"nat 2-radix representation = 100000000000000000000000000000000000000000001" : string

#sys_string_of_nat (#8, "nat 8-radix representation = ", nat, #0, #2, "");;
"nat 8-radix representation = 100000000001" : string

#sys_string_of_nat (#10, "nat 10-radix representation = ", nat, #0, #2, "");;
"nat 10-radix representation = 8589934593" : string

#sys_string_of_nat (#10, "nat first digit 10-radix representation = ",
# nat, #0, #1, "");;
"nat first digit 10-radix representation = 1" : string

#string_of_nat nat;;
"8589934593" : string

#string_for_read_of_nat nat;;
"#<8589934593>" : string

```

```
#sys_print_nat (#16, "nat 16-radix representation = ", nat, #0, #2, "");;
nat 16-radix representation = 200000001() : unit

#print_nat nat;;
8589934593() : unit

#print_nat_for_read nat;;
#<8589934593>() : unit

>(* Same remark as for sys_string_list_of_nat *)
#debug_string_vect_nat nat;;
[|"|00000002|00000001|"] : string vect

#debug_string_nat nat;;
"|00000002|00000001|" : string

#debug_print_nat nat;;
|00000002|00000001|() : unit
```

Special printing functions



```
sys_latex_print_nat†: int × string × nat × int × int × string → unit
latex_print_nat†: nat → unit
latex_print_for_read_nat†: nat → unit
sys_print_beautiful_nat†: int × string × nat × int × int × string → unit
print_beautiful_nat†: nat → unit
```

These special printing functions for natural numbers can be found into two libraries:

format_latex_numbers.ml: functions that split numbers into small lines so \LaTeX is able to handle them correctly,

format_numbers.ml: functions to present the `nat` as a numerical table according to common conventions.

`sys_latex_print_nat` (`base`, `before`, `nat`, `off`, `len`, `after`) prints `base` representation of subnat $nat_{off, len}$ between `before` and `after`, cutting line each `latex_margin` characters.

`latex_print_nat` and `latex_print_for_read_nat` are equivalent to:

```
let latex_print_nat nat =
  sys_latex_print_nat (#10, "", nat, #0,
                      num_digits_nat (nat, #0, length_nat nat), "")
;;

let latex_print_for_read_nat nat =
  sys_latex_print_nat (#10, "#<", nat, #0,
                      num_digits_nat (nat, #0, length_nat nat), ">")
;;
```

As an example this manual uses `latex_print_for_read_nat` as the printer for natural numbers.

`sys_print_beautiful_nat` (`base`, `before`, `nat`, `off`, `len`, `after`) prints base-radix representation of subnat $nat_{off,len}$ between `before` and `after` as a numerical table according to classical conventions.

`print_beautiful_nat` is equivalent to:

```
let print_beautiful_nat nat =
  sys_print_beautiful_nat (#10, "", nat, #0,
    num_digits_nat (nat, #0, length_nat nat), "")
;;
```

You can load the libraries with:

```
load_lib_file "format_latex_numbers";;
load_lib_file "format_numbers";;
```

Example

```
#let nat = make_nat #50;;
Value nat is #<0> : nat

#set_digit_nat (nat, #49, #1);;
() : unit

#print_beautiful_nat nat;;
10352 21304 67682 24942 50182 38987 58125 56497 26918 22829
15323 02783 39025 97280 29067 09060 12854 18454 96784 42287
66593 04257 74796 15304 02560 49867 14394 71495 42742 79767
67721 14810 59401 01347 02920 37191 58120 99936 58212 56286
17621 84379 22016 11343 67961 01316 26312 73818 64212 94556

73571 30891 00244 69518 17226 86663 79357 25012 27715 49253
01910 77163 01297 77340 98941 24750 20844 05834 13946 43737
89454 47786 92139 36390 56949 58460 67318 10387 73284 47182
89469 13683 23793 71211 71506 21872 25034 33123 21446 06272
33274 17368 32441 05465 856
() : unit
```

4.9 Logical operations on natural numbers



`land_digit_nat` : $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{unit}$

`lor_digit_nat` : $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{unit}$

`lxor_digit_nat` : $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{unit}$

`land_digit_nat` (`nat1`, `off1`, `nat2`, `off2`) performs “logical and” on digits $nat1_{off1}$ and $nat1_{off2}$, the resulting digit is put in $nat1_{off1}$.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
```

```
|00000003|00000002|00000001|00000000|() : unit

#land_digit_nat (nat, #3, nat, #1); debug_print_nat nat;;
|00000001|00000002|00000001|00000000|() : unit

#land_digit_nat (nat, #0, nat, #2); debug_print_nat nat;;
|00000001|00000002|00000001|00000000|() : unit
```

`lor_digit_nat (nat1, off1, nat2, off2)` performs “logical or” on digits $nat1_{off1}$ and $nat1_{off2}$, the resulting digit is put in $nat1_{off1}$.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#lor_digit_nat (nat, #3, nat, #1); debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#lor_digit_nat (nat, #0, nat, #2); debug_print_nat nat;;
|00000003|00000002|00000001|00000002|() : unit
```

`lxor_digit_nat (nat1, off1, nat2, off2)` performs “logical exclusive or” on digits $nat1_{off1}$ and $nat1_{off2}$, the resulting digit is put in $nat1_{off1}$.

Example

```
#let nat = create_nat #4;;
Value nat is #<0> : nat

#for i = #0 to #3 do set_digit_nat (nat, i, i) done; debug_print_nat nat;;
|00000003|00000002|00000001|00000000|() : unit

#lxor_digit_nat (nat, #3, nat, #1); debug_print_nat nat;;
|00000002|00000002|00000001|00000000|() : unit

#lxor_digit_nat (nat, #0, nat, #2); debug_print_nat nat;;
|00000002|00000002|00000001|00000002|() : unit
```

4.10 Miscellaneous power functions



`sqrt_nat`[†]: $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{nat}$
`power_base_int`[†]: $\text{int} \times \text{int} \rightarrow \text{nat}$
`power_base_nat`[†]: $\text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{nat}$

`sqrt_nat (nat, off, len)` computes the integer part of the square root of subnat $nat_{off, len}$.
`power_base_int (i, p)` computes i^p , where p is a positive `int`, and the resulting value is exact and a natural number.

`power_base_nat (base, nat, off, len)` computes `base` to $nat_{off, len}$ power.

Example

```
#let nat = nat_of_int #1729;;  
Value nat is #<1729> : nat
```

```
#sqrt_nat (nat, #0, #1);;  
#<41> : nat
```

```
#power_base_int (#10, #24);;  
#<1000000000000000000000000> : nat
```

```
#power_base_nat (#10, nat_of_int #24, #0, #1);;  
#<2003764205206896640> : nat
```


Chapitre 5

Big integers

Elements of type `big_int` are arbitrarily large signed numbers. All operations on this type are performed with copy update.

We have the following definition of type `big_int`:

```
type big_int = { Sign : int; Abs_Value : nat };;
```

In fact the sign is -1, 0 or 1 according to the classical conventions on signs as follows with generical writing:

```
let sign (x) =  
  when x<0 -> -1  
  | x=0 -> 0  
  | _ -> 1  
;;
```

You cannot create a `big_int` with a sign different from -1, 0 and 1. The sign is supposed to correspond to the real sign of the number and particularly that the equivalence

$$\text{sign}(x) = 0 \iff x = 0$$

is always respected. It is advisable not to introduce a `big_int` violating this convention by directly typing in such a record.

5.1 Creation of a `big_int`



`create_big_int`[†]: `int × nat → big_int`

We present here only the primitive function to create a `big_int`. You can of course create a `big_int` with coercion functions or using lexical convention.

`create_big_int (sign, nat)` verifies the sign condition and creates the corresponding `big_int`.

Example

```
#create_big_int (#1, nat_of_int #1);;  
#(1) : big_int
```



```
#create_big_int (#0, nat_of_int #1);

Evaluation Failed: failure "create_big_int"

#create_big_int (#-1, nat_of_int #1);
#(-1) : big_int

#create_big_int (#0, nat_of_int #0);
#(0) : big_int

#create_big_int (#1, nat_of_int #0);

Evaluation Failed: failure "create_big_int"
```

5.2 Arithmetic operations for big_ints

5.2.1 Incrementing big_ints



```
pred_big_int : big_int → big_int
succ_big_int : big_int → big_int
```

`pred_big_int n` (resp. `succ_big_int n`) is a primitive equivalent to $n-1$ (resp $n+1$).

Example

```
#succ_big_int (big_int_of_int biggest_int);
#(32768) : big_int

#succ_big_int (big_int_of_int monster_int);
#(-32767) : big_int

#pred_big_int (big_int_of_int least_int);
#(-32768) : big_int

#pred_big_int #(1.2e3);
#(1199) : big_int

#succ_big_int #(1234567890);
#(1234567891) : big_int
```

5.2.2 Arithmetic operations for big_ints



```
add_int_big_int†: int × big_int → big_int
add_big_int : big_int × big_int → big_int
minus_big_int : big_int → big_int
sub_big_int : big_int × big_int → big_int
mult_int_big_int†: int × big_int → big_int
mult_big_int : big_int × big_int → big_int
square_big_int†: big_int → big_int
```

quomod_big_int[†]: $\text{big_int} \times \text{big_int} \rightarrow \text{big_int} \times \text{big_int}$
quo_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$
div_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$
mod_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$
gcd_big_int[†]: $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$

`add_big_int`, `sub_big_int`, `mult_big_int` perform respectively the addition, subtraction and multiplication on `big_ints`.

`add_int_big_int` (*i*, *bi*) and `mult_int_big_int` (*i*, *bi*) perform respectively the addition and multiplication between the `int` *i* and the `big_int` *bi* and yield a `big_int`.

`minus_big_int` *bi* is the opposite of *bi*.

`square_big_int` *bi* computes the square of the `big_int` *bi* faster than multiplication in the manner of `square_nat`.

`quomod_big_int` (*bi1*, *bi2*) is the pair of the quotient and the remainder of the Euclidian division of *bi1* by *bi2*.

These two results may be consulted separately with `quo_big_int` (or `div_big_int` which is exactly the same function) and `mod_big_int` respectively.

`gcd_big_int` (*bi1*, *bi2*) computes the gcd of the two `big_ints` *bi1* and *bi2*. This gcd is always positive or null.

Example

```

#minus_big_int (big_int_of_int biggest_int);;
#(-32767) : big_int

#minus_big_int (big_int_of_int least_int);;
#(32767) : big_int

#minus_big_int (big_int_of_int monster_int);;
#(32768) : big_int

#mult_int_big_int (#2, big_int_of_int biggest_int);;
#(65534) : big_int

#quomod_big_int (#(123), #(3));;
#(41),#(0) : big_int * big_int

#quomod_big_int (#(123), #(-3));;
#(-41),#(0) : big_int * big_int

```

5.3 Comparisons between big_ints



eq_big_int[†]: $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$
compare_big_int[†]: $\text{big_int} \times \text{big_int} \rightarrow \text{int}$
lt_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$
le_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$
gt_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$
ge_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$

sign_big_int[†]: big_int → int
abs_big_int : big_int → big_int
max_big_int : big_int × big_int → big_int
min_big_int : big_int × big_int → big_int
zero_big_int[†]: big_int

eq_big_int, lt_big_int, le_big_int, gt_big_int, ge_big_int are arithmetic comparisons corresponding respectively to =, <, <=, >, >=.

The primitive function for these functions is compare_big_int, that is semantically equivalent to:

```
let compare_big_int (bigint1, bigint2) =
  when (bigint1 = bigint2) -> #0
  | (bigint1 < bigint2) -> #-1
  | _ -> #1;;
```

sign_big_int is completely equivalent to:

```
let sign_big_int bi = bi.Sign
;;
```

abs_big_int bi is the absolute value of the big_int bi.

max_big_int (bi1, bi2) and min_big_int (bi1, bi2) are respectively the greater and the smaller of the big_ints bi1 and bi2.

zero_big_int is the null big_int.

Example

```
#eq_big_int (succ_big_int (big_int_of_int biggest_int),
#          big_int_of_int monster_int);;
false : bool

#gt_big_int (succ_big_int (big_int_of_int biggest_int),
#          big_int_of_int biggest_int);;
true : bool

#sign_big_int (succ_big_int (big_int_of_int biggest_int));;
1 : int

#sign_big_int (big_int_of_int monster_int);;
-1 : int

#abs_big_int (big_int_of_int monster_int);;
#(32768) : big_int
```

5.4 Coercion functions on big_ints

5.4.1 Coercion between ints and big_ints



is_int_big_int[†]: `big_int` → `bool`
int_of_big_int[†]: `big_int` → `int`
big_int_of_int[†]: `int` → `big_int`

`is_int_big_int bi` yields true if and only if the `big_int` `bi` fits in an `int`.
`int_of_big_int bi` is the `int` representing the `big_int` `bi` if `bi` fits in an `int`.
`big_int_of_int` maps an `int` onto the corresponding `big_int`.

Example

```
#is_int_big_int (big_int_of_int biggest_int);;
true : bool

#is_int_big_int (big_int_of_int monster_int);;
true : bool

#is_int_big_int (abs_big_int (big_int_of_int monster_int));;
false : bool

#is_int_big_int (succ_big_int (big_int_of_int biggest_int));;
false : bool

#int_of_big_int (big_int_of_int biggest_int);;
32767 : int

#int_of_big_int (succ_big_int (big_int_of_int biggest_int));;

Evaluation Failed: failure "int_of_big_int"

#big_int_of_int monster_int;;
#(-32768) : big_int
```

5.4.2 Coercion between floating point numbers and big_int

float_of_big_int[†]: `big_int` → `float`
big_int_of_float[†]: `float` → `big_int`

`float_of_big_int` and `big_int_of_float` map elements of type `big_int` into elements of type `float`.

Example

```
#float_of_big_int #(1.23456789e8);;
123456800.0 : float

#big_int_of_float it;;
#(123456800) : big_int
```

5.4.3 Coercion between nats and big_ints



`big_int_of_nat`[†]: `nat` → `big_int`
`nat_of_big_int`[†]: `big_int` → `nat`

`big_int_of_nat` maps naturally a `nat` onto the corresponding `big_int`.
`nat_of_big_int bi` maps `big_int bi` onto the corresponding `nat` if `bi` is positive.
`is_big_int_nat` can be written as :

```
#let is_big_int_nat bi = ge_int (sign_big_int bi, #0);;
Value is_big_int_nat is <fun> : big_int -> bool
```

Example

```
#big_int_of_nat (nat_of_string "123456789");;
#(123456789) : big_int
```

```
#nat_of_big_int #(1.2e3);;
#<1200> : nat
```

```
#nat_of_big_int #(-1.2e3);;
```

```
Evaluation Failed: failure "nat_of_big_int"
```

```
#is_big_int_nat #(1.2e3);;
true : bool
```

```
#is_big_int_nat #(-1.2e3);;
false : bool
```

5.4.4 Coercion between strings and `big_int`

Reading `big_int`s



`simple_big_int_of_string`[†]: `int` × `string` × `int` × `int` → `big_int`
`decimal_of_string`[†]: `int` × `string` × `int` × `int` → `string` × `int`
`sys_big_int_of_string`[†]: `int` × `string` × `int` × `int` → `big_int`
`big_int_of_string`[†]: `string` → `big_int`

CAML is able to read `big_int` entered directly, according to the following lexical convention:

```
Separator ::= ' ' | '\f' | '\n' | '\r' | '\t' | '\\'  

Pseudodigit ::= Digit | Separator  

Decimal ::= {'-' | '+'} Separator* Digit Pseudodigit*  

           {'.' Separator* Digit Pseudodigit*} {'e' {'-' | '+'} INT}  

BIG_INT ::= '#(' Decimal ')'
```

Example

```
##(12);;
#(12) : big_int
```

```
##(12.0);;
#(12) : big_int

##(1.2e1);;
#(12) : big_int

##(120e-1);;
#(12) : big_int
```

`simple_big_int_of_string` recognizes simple expressions for `big_int`, i.e. without mantissa and without floating point.

`decimal_of_string` is the fundamental function for recognizing numerical expressions with floating point and mantissa. `decimal_of_string (base, s, off, len)` yields the pair of a string representing a simple `big_int` and an `int` representing the corresponding mantissa. Roughly speaking, if $s_{off, len}$ is of the form `s1.s2es3` then the result is `(s1^s2, s3-length_string s2)`.

`sys_big_int_of_string (base, s, off, len)` maps the substring $s_{off, len}$ in base `base` onto a `big_int`.

`big_int_of_string s` is equivalent to:

```
let big_int_of_string s =
  sys_big_int_of_string (#10, s, #0, length_string s)
;;
```

Example

```
#simple_big_int_of_string (#10, "1200", #0, #4);;
#(1200) : big_int

#simple_big_int_of_string (#10, "1.2e3", #0, #5);;

Evaluation Failed: failure "base_digit_of_char"

#decimal_of_string (#10, "1200", #0, #4);;
("1200",0) : string * int

#decimal_of_string (#10, "1.2e3", #0, #5);;
("12",2) : string * int

#decimal_of_string (#10, "1.2e-1", #0, #6);;
("12",-2) : string * int

#sys_big_int_of_string (#16, "FF", #0, #2);;
#(255) : big_int

#big_int_of_string "1.2e3";;
#(1200) : big_int
```

Printing big_ints



```

sys_string_of_big_int : int × string × big_int × string → string
string_of_big_int†: big_int → string
string_for_read_of_big_int†: big_int → string
sys_print_big_int†: int × string × big_int × string → unit
print_big_int†: big_int → unit
print_big_int_for_read : big_int → unit

```

`sys_string_of_big_int` (`base`, `before`, `bi`, `after`) writes into the resulting string first the string `before`, then the representation of the `big_int` `bi` in the base `base`, and finally the string `after`.

This function is equivalent to:

```

let sys_sys_string_of_big_int (base, before, bi, after) =
  sys_string_of_nat (base,
                    (if eq_int (bi.Sign, #-1)
                     then (before ^ "-") else before),
                    bi.Abs_Value, #0, num_digits_big_int bi, after)
;;

```

`string_of_big_int` `bi` is equivalent to:

```

let string_of_big_int bi =
  sys_string_of_big_int (#10, "", bi, "")
;;

```

`string_for_read_of_big_int` `bi` is analogous to `string_of_big_int` `bi` except that the number is written between `#(` and `)`, according to conventions to write `big_int`s.

It is equivalent to:

```

let string_for_read_of_big_int bi =
  sys_string_of_big_int (#10, "#(", bi, ")")
;;

```

`sys_print_of_big_int` (`base`, `before`, `bi`, `after`) prints the `base`-radix representation of the `big_int` `bi` between strings `before` and `after`.

`print_big_int` and `print_big_int_for_read` `nat` are equivalent to:

```

let print_big_int bi = sys_print_big_int (#10, "", bi, "");;
let print_big_int_for_read bi = sys_print_big_int (#10, "#(", bi, ")");;

```

Example

```

#sys_string_of_big_int
# (#16, "big_int 16-radix representation = ", #(1.2e3), "");;
"big_int 16-radix representation = 4B0" : string

#sys_string_of_big_int (#2, "big_int 2-radix representation = ", #(1.2e3), "");;
"big_int 2-radix representation = 10010110000" : string

#sys_string_of_big_int (#8, "big_int 8-radix representation = ", #(1.2e3), "");;

```

```

"big_int 8-radix representation = 2260" : string

#sys_string_of_big_int
# (#10, "big_int 10-radix representation = ", #(1.2e3), "");
"big_int 10-radix representation = 1200" : string

#string_of_big_int #(1.2e3);
"1200" : string

#string_for_read_of_big_int #(1.2e3);
"#(1200)" : string

#sys_print_big_int (#16, "big_int 16-radix representation = ", #(1.2e3), "");
big_int 16-radix representation = 4B0() : unit

#print_big_int #(1.2e3);
1200() : unit

#print_big_int_for_read #(1.2e3);
#(1200)() : unit

```

Special printing functions



```

sys_latex_print_big_int†: int × string × big_int × string → unit
latex_print_big_int†: big_int → unit
latex_print_for_read_big_int†: big_int → unit
sys_print_beautiful_big_int†: α × string × big_int × string → unit
print_beautiful_big_int†: big_int → unit

```

These special printing functions for `big_ints` can be found into two libraries:

format_latex_numbers.ml: functions that split numbers into small lines so \LaTeX is able to handle them correctly,

format_numbers.ml: functions to present the `big_int` as a numerical table according to common conventions.

`sys_latex_print_big_int` (`base`, `before`, `bi`, `after`) prints base representation of the `big_int` `bi` between `before` and `after`, cutting line each `latex_margin` characters.

`latex_print_big_int` and `latex_print_for_read_big_int` are equivalent to:

```

let latex_print_big_int bi =
  sys_latex_print_big_int (#10, "", bi, "")

and latex_print_for_read_big_int bi =
  sys_latex_print_big_int (#10, "#(", bi, ")")
;;

```

As an example this manual uses `latex_print_for_read_big_int` as the printer for `big_ints`.

`sys_print_beautiful_big_int` (`base`, `before`, `bi`, `after`) prints base-radix representation of the `big_int` `bi` between `before` and `after` as a numerical table according to classical conventions.

`print_beautiful_big_int` is equivalent to:

```
let print_beautiful_big_int bi =
  sys_print_beautiful_big_int (#10, "", bi, "")
;;
```

You can load the libraries with:

```
load_lib_file "format_latex_numbers";;
load_lib_file "format_numbers";;
```

Example

```
#print_beautiful_big_int #(1e312);;
10000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000

00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 000
() : unit
```

5.5 Miscellaneous information functions on `big_int`s



`nth_digit_big_int`[†]: `big_int` × `big_int` → `int`
`leading_digit_big_int`[†]: `big_int` → `int`
`num_digits_big_int`[†]: `big_int` → `int`

`nth_digit_big_int` (`bi`, `n`) returns the n -th 2^{32} -digit of the absolute value of `bi`.

`leading_digit_big_int` `bi` is the leading digit of the 2^{32} -radix representation of the `big_int` `bi`.

`num_digits_big_int` `bi` is the length of the 2^{32} -radix representation of the `big_int` `bi`.

5.6 Miscellaneous power functions



`sqrt_big_int`[†]: `big_int` → `big_int`
`power_int_positive_int`[†]: `int` × `int` → `big_int`
`power_int_positive_big_int`[†]: `int` × `big_int` → `big_int`
`base_power_big_int`[†]: `int` × `int` × `big_int` → `big_int`
`power_big_int_positive_int`[†]: `big_int` × `int` → `big_int`
`power_big_int_positive_big_int`[†]: `big_int` × `big_int` → `big_int`

`sqrt_big_int bi` computes the integer part of the square root of the `big_int` `bi`.

`power_int_positive_int (i1, i2)` yields the `big_int` resulting from the computation of $i1^{i2}$ where $i1$ and $i2$ are ints and $i2 \geq 0$.

`power_int_positive_big_int (i, bi)` yields the `big_int` resulting from the computation of i^{bi} where bi is a positive `big_int`.

`base_power_big_int (base, n, bi)` computes $bi \times base^n$ as far as it is an integer.

`power_big_int_positive_int (bi, i)` yields the `big_int` resulting from the computation of bi^i where i is a positive int.

`power_big_int_positive_big_int (bi1, bi2)` yields the `big_int` resulting from the computation of $bi1^{bi2}$ where $bi2$ is a positive `big_int`.

Example

```
#sqrt_big_int #(1e10);
#(100000) : big_int
```

```
#power_int_positive_int (#10, #11);
#(100000000000) : big_int
```

```
#power_int_positive_big_int (#10, #(11));
#(1215752192) : big_int
```

```
#base_power_big_int (#10, #2, #(12));
#(1200) : big_int
```

```
#base_power_big_int (#10, #-2, #(1.2e3));
```

```
Evaluation Failed: failure "base_power_big_int"
```

```
#base_power_big_int (#10, #-1, #(12));
```

```
Evaluation Failed: failure "base_power_big_int"
```

```
#power_big_int_positive_int (#(10), #11);
#(100000000000) : big_int
```

```
#power_big_int_positive_big_int (#(10), #(11));
#(100000000) : big_int
```


Chapitre 6

Rational numbers of type `ratio`

Elements of type `ratio` are arbitrary large rational numbers. They are internally represented with a couple of `big_ints`, the second one being a positive integer. *infinity* (`1/0` or `-1/0`) and *undefined* (`0/0`) exist as natural values of type `ratio` on request.

We have the following definition of type `ratio`:

```
type ratio = { Numerator : big_int; Denominator : big_int; Normalized : bool };;
```

The denominator is supposed to be positive and the sign of an element of type `ratio` is supposed to be the sign of its numerator. It is advisable not to introduce a value of type `ratio` violating this convention by typing in such a record directly.

6.1 Normalization of ratios



```
normalize_ratio†: ratio → ratio  
get_normalize_ratio†: unit → bool  
set_normalize_ratio†: bool → bool  
cautious_normalize_ratio†: ratio → ratio  
get_normalize_ratio_when_printing†: unit → bool  
set_normalize_ratio_when_printing†: bool → bool  
cautious_normalize_ratio_when_printing†: ratio → ratio  
is_normalized_ratio†: ratio → bool
```

Normalization is adaptable. Two flags allow you to choose when you want to normalize rational numbers: after each computation, only before printing, or never. You can set or consult them with the `(g|s)et_normalize_ratio{when_printing}` functions. For more information see chapter 8.

To force the normalization of a particular rational number, use the normalization primitive `normalize_ratio`.

On the other hand, if you want the normalization to be performed only if one of these flags is turned on, use `cautious_normalize_ratio` or `cautious_normalize_ratio_when_printing`.

When the `Normalized` label of a `ratio` is set to `true`, normalization is not performed anymore on this number. So the gcd of the numerator and denominator of a rational number is computed at the most once. `is_normalized_ratio r` tells you if the rational number `r` is already normalized.

6.2 Creation of a ratio



create_ratio[†]: `big_int × big_int → ratio`
create_normalized_ratio[†]: `big_int × big_int → ratio`

We present here only the primitive functions to create a number of type `ratio`: you can of course create a `ratio` with coercion functions or using lexical convention.

`create_ratio (num, den)` creates the `ratio` with numerator (resp. denominator) equal to `num` (resp. `den`), set up the sign of the rational number, verifies if its denominator is not null if necessary, normalizes this rational if normalization during computing is requested and sets `Normalized` label with the value of this normalization flag.

`create_normalized_ratio (num, den)` creates the `ratio` with numerator (resp. denominator) equal to `num` (resp. `den`), reports the eventual sign of `den` on the numerator of the created `ratio`, verifies if denominator is not null when null denominators are prohibited and sets `Normalized` label with `true`. It is interesting to use this creation function when you are sure a rational number to be already normalized, since this `ratio` will be no more normalized, but off course you must be right.

By default null denominators are prohibited. For more information see chapter 8.

Example

```
#create_ratio (#(2), #(3));
#[2/3] : ratio
```

```
#create_ratio (#(2), #(-3));
#[-2/3] : ratio
```

```
#create_ratio (#(4), #(6));
#[2/3] : ratio
```

```
#create_ratio (#(1), #(0));
```

```
Evaluation Failed: failure "create_ratio infinite or undefined rational number"
```

```
#create_normalized_ratio (#(4), #(-6));
#[-4/6] : ratio
```

6.3 Miscellaneous information functions on ratios



numerator_ratio[†]: `ratio → big_int`
denominator_ratio[†]: `ratio → big_int`
null_denominator[†]: `ratio → bool`
verify_null_denominator[†]: `ratio → bool`

`numerator_ratio` and `denominator_ratio` consult the corresponding label of the record.

`null_denominator r` and `verify_null_denominator` test if the denominator of the `ratio` is null. `verify_null_denominator` fails when a null denominator is encountered and null denominators are prohibited.

Example

```
#null_denominator {Numerator=#(1);Denominator=#(0);Normalized=false}};
true : bool

#verify_null_denominator {Numerator=#(1);Denominator=#(0);Normalized=false}};

Evaluation Failed: failure "infinite or undefined rational number"
```

6.4 Arithmetic operations for ratios



```
add_int_ratio†: int × ratio → ratio
add_big_int_ratio†: big_int × ratio → ratio
add_ratio : ratio × ratio → ratio
minus_ratio : ratio → ratio
sub_ratio : ratio × ratio → ratio
mult_int_ratio†: int × ratio → ratio
mult_big_int_ratio†: big_int × ratio → ratio
mult_ratio : ratio × ratio → ratio
square_ratio†: ratio → ratio
inverse_ratio†: ratio → ratio
div_int_ratio†: int × ratio → ratio
div_big_int_ratio†: big_int × ratio → ratio
div_ratio_int†: ratio × int → ratio
div_ratio_big_int†: ratio × big_int → ratio
div_ratio : ratio × ratio → ratio
```

`add_ratio` adds two rational numbers.

`add_int_ratio` (resp. `add_big_int_ratio`) performs the addition of an `int` (resp. `big_int`) to a rational number. These functions are semantically equivalent to:

```
let add_int_ratio (i, r) = add_ratio (ratio_of_int i, r)
and add_big_int_ratio (bi, r) = add_ratio (ratio_of_big_int bi, r)
;;
```

`minus_ratio r` is the opposite of the rational number r .

`sub_ratio` is the subtraction between rational numbers.

`mult_ratio` multiplies two rational numbers.

`mult_int_ratio` (resp. `mult_big_int_ratio`) performs the multiplication of an `int` (resp. `big_int`) to a rational number. These functions are semantically equivalent to:

```
let mult_int_ratio (i, r) = mult_ratio (ratio_of_int i, r)
and mult_big_int_ratio (bi, r) = mult_ratio (ratio_of_big_int bi, r)
;;
```

`square_ratio r` computes the square of the rational number r . It is equivalent to:

```
let square_ratio r =
  { Numerator = square_big_int r.Numerator;
    Denominator = square_big_int r.Denominator;
    Normalized = r.Normalized }
;;
```

`inverse_ratio` is equivalent to:

```
let inverse_ratio r =
  { Numerator = r.Denominator;
    Denominator = r.Numerator;
    Normalized = r.Normalized }
;;
```

`div_ratio` performs the division on rational numbers.

`div_int_ratio` (resp. `div_big_int_ratio`) performs the division of an `int` (resp. `big_int`) by a rational number. These functions are semantically equivalent to:

```
let div_int_ratio (i, r) = div_ratio (ratio_of_int i, r)
and div_big_int_ratio (bi, r) = div_ratio (ratio_of_big_int bi, r)
;;
```

`div_ratio_int` (resp. `div_ratio_big_int`) performs the division of a rational number by an `int` (resp. `big_int`). These functions are semantically equivalent to:

```
let div_ratio_int (r, i) = div_ratio (r, ratio_of_int i)
and div_ratio_big_int (r, bi) = div_ratio (r, ratio_of_big_int bi)
;;
```

Example

```
#add_ratio (#[1/2], #[1/2]);;
#[1/1] : ratio

#sub_ratio (#[1/2], #[1/2]);;
#[0/1] : ratio

#inverse_ratio #[1/2];;
#[2/1] : ratio

#mult_int_ratio (#2, #[1/2]);;
#[1/1] : ratio

#square_ratio #[1/3];;
#[1/9] : ratio

#div_int_ratio (#2, #[2]);;
#[1/1] : ratio
```

6.5 Rounding ratios



is_integer_ratio[†]: `ratio` \rightarrow `bool`
integer_ratio[†]: `ratio` \rightarrow `big_int`
floor_ratio[†]: `ratio` \rightarrow `big_int`
round_ratio[†]: `ratio` \rightarrow `big_int`
ceiling_ratio[†]: `ratio` \rightarrow `big_int`

`is_integer_ratio` normalizes its argument, in order to determine if it is an integer. It is equivalent to:

```
let is_integer_ratio r =
  eq_big_int ((normalize_ratio r).Denominator, unit_big_int)
;;
```

`integer_ratio` is the odd function of truncature for rational numbers.

`floor_ratio` (resp. `round_ratio`, `ceiling_ratio`) are the mathematical functions for rounding to the lower (resp. nearest, higher) integer.

Example

```
#is_integer_ratio #[1.2e3/1200];;
true : bool
```

```
#integer_ratio #[1/3];;
#(0) : big_int
```

```
#floor_ratio #[1/3];;
#(0) : big_int
```

```
#round_ratio #[1/3];;
#(0) : big_int
```

```
#ceiling_ratio #[1/3];;
#(1) : big_int
```

```
#integer_ratio #[1/2];;
#(0) : big_int
```

```
#floor_ratio #[1/2];;
#(0) : big_int
```

```
#round_ratio #[1/2];;
#(1) : big_int
```

```
#ceiling_ratio #[1/2];;
#(1) : big_int
```

```
#integer_ratio #[-1/3];;
#(0) : big_int
```

```
#floor_ratio #[-1/3];;
#(-1) : big_int
```

```
#round_ratio #[-1/3];;
#(0) : big_int
```

```
#ceiling_ratio #[-1/3];;
#(0) : big_int
```

```
#integer_ratio #[-1/2];;
#(0) : big_int
```



```
#floor_ratio #[-1/2];;
#(-1) : big_int

#round_ratio #[-1/2];;
#(-1) : big_int

#ceiling_ratio #[-1/2];;
#(0) : big_int
```

6.6 Comparisons between and with ratios



```
eq_big_int_ratio†: big_int × ratio → bool
compare_big_int_ratio†: big_int × ratio → int
lt_big_int_ratio†: big_int × ratio → bool
le_big_int_ratio†: big_int × ratio → bool
gt_big_int_ratio†: big_int × ratio → bool
ge_big_int_ratio†: big_int × ratio → bool
eq_ratio†: ratio × ratio → bool
compare_ratio†: ratio × ratio → int
lt_ratio : ratio × ratio → bool
le_ratio : ratio × ratio → bool
gt_ratio : ratio × ratio → bool
ge_ratio : ratio × ratio → bool
sign_ratio†: ratio → int
report_sign_ratio†: ratio × big_int → big_int
abs_ratio : ratio → ratio
min_ratio : ratio × ratio → ratio
max_ratio : ratio × ratio → ratio
```

`lt_ratio`, `le_ratio`, `gt_ratio`, `ge_ratio` (resp. `lt_big_int_ratio`, `le_big_int_ratio`, `gt_big_int_ratio`, `ge_big_int_ratio`) are arithmetic comparisons corresponding respectively to `<`, `<=`, `>`, `>=` between elements of type `ratio` (resp. element of type `big_int` and `ratio`).

The primitive function for these functions is `compare_ratio` (resp. `compare_big_int_ratio`), that is semantically equivalent to:

```
let compare_ratio (r1, r2) =
  when (r1 = r2) -> #0
  | (r1 < r2) -> #-1
  | _ -> #1

and compare_big_int_ratio (bi, r) =
  when (bi = r) -> #0
  | (bi < r) -> #-1
  | _ -> #1
;;
```

`sign_ratio` is equivalent to:

```
let sign_ratio r = sign_big_int r.Numerator ;;
```

`report_sign_ratio (bi, r)` transfers the sign of the `big_int` `bi` on the rational number `r`.

`abs_ratio r` is the absolute value of the rational number `r`.

`max_ratio (r1, r2)` and `min_ratio (r1, r2)` are respectively the greater one and the smaller one of rational numbers `r1` and `r2`.

Example

```
#compare_ratio ([1.2e3],[12e2]);;
```

```
0 : int
```

```
#compare_big_int_ratio ((1.2e3),#[1200/1]);;
```

```
0 : int
```

```
#compare_big_int_ratio ((1.2e3),#[12e2]);;
```

```
0 : int
```

6.7 Coercion functions on ratios

6.7.1 Coercion between ints and ratios



`int_of_ratio†`: `ratio` \rightarrow `int`

`ratio_of_int†`: `int` \rightarrow `ratio`

`int_of_ratio r` is the `int` representing the rational number `r` if `r` fits in an `int`, otherwise `int_of_ratio` fails.

`ratio_of_int` maps naturally an `int` onto the corresponding `ratio`.

Example

```
#int_of_ratio [1.2e3];;
```

```
1200 : int
```

```
#int_of_ratio (add_int_ratio (#1, ratio_of_int biggest_int));;
```

```
Evaluation Failed: failure "int_of_ratio"
```

```
#int_of_ratio (add_int_ratio (#-1, ratio_of_int least_int));;
```

```
-32768 : int
```

```
#ratio_of_int #1;;
```

```
#[1/1] : ratio
```

6.7.2 Coercion between nats and ratios



`nat_of_ratio†`: `ratio` \rightarrow `nat`

`ratio_of_nat†`: `nat` \rightarrow `ratio`

`nat_of_ratio r` maps rational number `r` onto the corresponding `nat`, if `r` is an integer and positive.

`ratio_of_nat` maps naturally a `nat` onto the corresponding rational number.

Example

```
#nat_of_ratio #[1.2e3];;
#<1200> : nat
```

```
#nat_of_ratio #[-1.2e3];;
```

```
Evaluation Failed: failure "nat_of_ratio"
```

```
#nat_of_ratio #[1.2];;
```

```
Evaluation Failed: failure "nat_of_ratio"
```

```
#nat_of_ratio #[1/2];;
```

```
Evaluation Failed: failure "nat_of_ratio"
```

6.7.3 Coercion between `big_ints` and ratios



`big_int_of_ratio`[†]: `ratio` → `big_int`

`ratio_of_big_int`[†]: `big_int` → `ratio`

`big_int_of_ratio r` maps rational number `r` onto the corresponding `big_int`, if `r` is an integer.

`ratio_of_big_int` maps naturally a `big_int` onto the corresponding rational number.

Example

```
#big_int_of_ratio #[1.2];;
```

```
Evaluation Failed: failure "big_int_of_ratio"
```

6.7.4 Coercion between floating point numbers and ratios



`float_of_ratio`[†]: `ratio` → `float`

`ratio_of_float`[†]: `float` → `ratio`

`float_of_ratio` and `ratio_of_float` map elements of type `ratio` into elements of type `float`.

Example

```
#float_of_ratio #[1.23456789e8];;
123456800.0 : float
```

```
#ratio_of_float it;;
#[123456800/1] : ratio
```

6.7.5 Coercion between strings and ratios

Reading ratios



sys_ratio_of_string[†]: int × string × int × int → ratio
ratio_of_string[†]: string → ratio

CAML is able to read numbers of type `ratio` entered directly, according to the following lexical convention:

```
Separator ::= ' ' | '\f' | '\n' | '\r' | '\t' | '\\'  

Pseudodigit ::= Digit | Separator  

Decimal ::= {'-' | '+'} Separator* Digit Pseudodigit*  

           {'.' Separator* Digit Pseudodigit*} {'e' {'-' | '+'} INT}  

RATIO ::= '#[' Decimal {'/' Decimal} ']'
```

Example

```
##[12];;  
#[12/1] : ratio  
  
##[12.0];;  
#[12/1] : ratio  
  
##[1.2e1];;  
#[12/1] : ratio  
  
##[120e-1];;  
#[12/1] : ratio  
  
##[12/1];;  
#[12/1] : ratio  
  
##[60/5];;  
#[12/1] : ratio  
  
##[1.2/1e-1];;  
#[12/1] : ratio  
  
##[120/10];;  
#[12/1] : ratio
```

`sys_ratio_of_string (base, s, off, len)` maps the substring $s_{off, len}$ in base `base` onto a element of type `ratio`.

`ratio_of_string s` is equivalent to:

```
let ratio_of_string s =  
  sys_ratio_of_string (#10, s, #0, length_string s)  
;;
```

Example

```
#sys_ratio_of_string (#16, "FF", #0, #2);;  
#[255/1] : ratio  
  
#ratio_of_string "1.2e3/5e1";;  
#[24/1] : ratio
```

Printing ratios



```

sys_string_of_ratio†: int × string × ratio × string → string
string_of_ratio†: ratio → string
string_for_read_of_ratio†: ratio → string
sys_print_ratio†: int × string × ratio × string → unit
print_ratio†: ratio → unit
print_ratio_for_read†: ratio → unit

```

`sys_string_of_ratio` (`base`, `before`, `r`, `after`) returns the concatenation of the string `before`, the string representation of rational number `r` in base `base`, and finally the string `after`. In fact if the approximation when printing rational numbers is turned on, the result is the floating point approximation to the floating point precision of the rational number. This is not implemented for `base` \neq 10. For more information see section 6.9 and chapter 8.

`string_of_ratio` `r` is equivalent to:

```

let string_of_ratio r =
  sys_string_of_ratio (#10, "", r, "")
;;

```

`string_for_read_of_ratio` `r` is analogous to `string_of_ratio` `r` except that the number is written between `#[` and `]`, according to conventions to write “big” numbers.

It is equivalent to:

```

let string_for_read_of_ratio r =
  sys_string_of_ratio (#10, "#[" , r, "]" )
;;

```

`sys_print_of_ratio` (`base`, `before`, `r`, `after`) prints the base-radix representation of the rational number `r` between strings `before` and `after`. It is equivalent to:

```

let sys_print_ratio (base, before, r, after) =
  print_string (sys_string_of_ratio (base, before, r, after))
;;

```

`print_ratio` and `print_ratio_for_read` are equivalent to:

```

let print_ratio r = sys_print_ratio (#10, "", r, "");;
let print_ratio_for_read r = sys_print_ratio (#10, "#[" , r, "]" );;

```

Example

```

#sys_string_of_ratio
# (#16, "ratio 16-radix representation = ", #[1.2e3/1.1e2], "");;
"ratio 16-radix representation = 78/B" : string

#sys_string_of_ratio
# (#2, "ratio 2-radix representation = ", #[1.2e3/1.1e2], "");;
"ratio 2-radix representation = 1111000/1011" : string

#sys_string_of_ratio

```

```
# (#8, "ratio 8-radix representation = ", #[1.2e3/1.1e2], "");;
"ratio 8-radix representation = 170/13" : string

#sys_string_of_ratio
# (#10, "ratio 10-radix representation = ", #[1.2e3/1.1e2], "");;
"ratio 10-radix representation = 120/11" : string

#string_of_ratio #[1.2e3/1.1e2];;
"120/11" : string

#string_for_read_of_ratio #[1.2e3/1.1e2];;
"#[120/11]" : string

#sys_print_ratio (#16, "ratio 16-radix representation = ", #[1.2e3/1.1e2], "");;
ratio 16-radix representation = 78/B() : unit

#print_ratio #[1.2e3/1.1e2];;
120/11() : unit

#print_ratio_for_read #[1.2e3/1.1e2];;
#[120/11]() : unit
```

Special printing functions



```
sys_latex_print_ratio†: int × string × ratio × string → unit
latex_print_ratio†: ratio → unit
latex_print_for_read_ratio†: ratio → unit
sys_print_beautiful_ratio†: int × string × ratio × string → unit
print_beautiful_ratio†: ratio → unit
```

These special printing functions for rational numbers can be found into two libraries:

format_latex_numbers.ml: functions that split numbers into small lines so \LaTeX is able to handle them correctly,

format_numbers.ml: functions to present the rational number as a numerical table according to common conventions.

`sys_latex_print_ratio (base, before, r, after)` prints `base` representation of the rational number `r` between `before` and `after`, cutting line each `latex_margin` characters.

`latex_print_ratio` and `latex_print_for_read_ratio` are equivalent to:

```
let latex_print_ratio r = sys_latex_print_ratio (#10, "", r, "")
```

```
and latex_print_for_read_ratio r = sys_latex_print_ratio (#10, "#[", r, "]" )
;;
```

As an example this manual uses `latex_print_for_read_ratio` as the printer for rational numbers.

`sys_print_beautiful_ratio (base, before, r, after)` prints `base`-radix representation of the rational number `r` between `before` and `after` as a numerical table according to classical conventions.

`print_beautiful_ratio` is equivalent to:

```
let print_beautiful_ratio r =
  sys_print_beautiful_ratio (#10, "", r, "")
;;
```

You can load the libraries with:

```
load_lib_file "format_latex_numbers";;
load_lib_file "format_numbers";;
```

Example

```
#print_beautiful_ratio #[1.2e3/1.1e2];;
120/11() : unit

#print_beautiful_ratio #[12345e105/1.1e2];;
12345 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 0000
/11() : unit

#print_beautiful_ratio #[12345e1/1.1e201];;
2469/
22000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
() : unit
```

6.8 Miscellaneous power functions



`power_ratio_positive_int`[†]: $\text{ratio} \times \text{int} \rightarrow \text{ratio}$
`power_ratio_positive_big_int`[†]: $\text{ratio} \times \text{big_int} \rightarrow \text{ratio}$

`power_ratio_positive_int` (*r*, *i*) yields the rational number resulting from the computation of r^i where *i* is a positive `int`.

`power_ratio_positive_big_int` (*r*, *bi*) yields the rational number resulting from the computation of r^{bi} where *r* is a positive rational number.

Example

```
#power_ratio_positive_int ([1.09],#10);;
#[236736367459211723401/10000000000000000000] : ratio

#power_ratio_positive_big_int ([2], #(32));;
#[4294967296/1] : ratio
```

6.9 Floating point approximations of rational numbers



msd_ratio[†]: ratio → int
round_futur_last_digit[†]: string × int × int → bool
approx_ratio_fix[†]: int × ratio → string
approx_ratio_exp[†]: int × ratio → string
float_of_rational_string[†]: ratio → string

msd_ratio *r* is the position of the leading digit of the decimal expansion of a strictly positive rational number. If

$$r = \sum_{k=-p}^{k=N} r_k 10^k$$

then we have msd_ratio *r* = *N*.

round_futur_last_digit (*s*, *off*, *len*) rounds off in place the last but one digit-character of the substring *s_{off,len}* and yields true if and only if it causes an overflow, i.e. if *s* contains an integer of the form $10^n - 1$. In this case after transformation *s* will contain only 0 characters and the resulting value of the function is true.

approx_ratio_fix (*n*, *r*) approaches the rational number *r* with a *n* 10-digits fix point representation. This is an odd function and the last digit is round off. In an informal manner, the printing format is:

`integer_part . decimal_part_with_n_digits`

approx_ratio_exp (*n*, *r*) approaches the rational number *r* with a *n* 10-digits floating point representation. This is an odd function and the last digit is round off. In an informal manner, the printing format is:

`(+|-) (0. n_first_digits e msd) | (1. n_zeros e (msd+1))`

float_of_rational_string is the same function with a default precision value according to the floating point precision flag. For more information see chapter 8.

Example

```

#msd_ratio #[1.2e3/1.1e2];;
1 : int

#let s = "+109090909090909";;
Value s is "+109090909090909" : string

#round_futur_last_digit (s, #0, #16);;
false : bool

#s;;
"+109090909090919" : string

#approx_ratio_fix (#12, #[1.2e3/1.1e2]);;
"+10.909090909091" : string

#approx_ratio_fix (#-1, #[1.2e3/1.1e2]);;
"+1" : string

```



```
#approx_ratio_exp (#12, #[1.2e3/1.1e2]);  
"+0.109090909091e2" : string  
  
#approx_ratio_exp (#2, #[1.2e3/1.1e2]);  
"+0.11e2" : string  
  
#float_of_rational_string #[1.2e3/1.1e2];  
"0.109090909091e2" : string
```

Chapitre 7

Rational numbers of type num

Elements of type `num` are arbitrarily large rational numbers, with optimization of the representation. We have the following definition of type `num`:

```
type num = Int of int | Big_int of big_int | Ratio of ratio
;;
```

7.1 Normalization of nums



`normalize_num`[†]: `num` → `num`
`cautious_normalize_num_when_printing`[†]: `num` → `num`

`normalize_num n` optimizes the representation of `n`. A rational will be normalized and if possible coerced to an integer. A “big” integer will be mapped onto a “small” one if possible.

`cautious_normalize_num_when_printing n` normalizes `n` in the above sense when the normalization when printing flag is turned on. For more information see chapter 8.

Beware:

→ It is supposed that “big” integers do not fit in a “small” one. You can introduce a “small” integer with constructor `Big_int`, this will not cause any error but it will be less efficient.

7.2 Miscellaneous information functions on nums



`numerator_num`[†]: `num` → `num`
`denominator_num`[†]: `num` → `num`

`numerator_num` and `denominator_num` are respectively the numerator and the denominator of the normalized representation of its argument.

Example

```
#numerator_num #{4/6};;
#{2} : num
```

```
#denominator_num #{4/6};;
#{3} : num
```

7.3 Arithmetic operations for nums

7.3.1 Incrementing and decrementing

Incrementing and decrementing numbers



```
pred_num : num → num
succ_num : num → num
```

`pred_num n` (resp. `succ_num n`) is a primitive equivalent to `n-1` (resp `n+1`).

Example

```
#pred_num #{4/6};;
#{-1/3} : num

#succ_num #{4/6};;
#{5/3} : num
```

Incrementing and decrementing counters



```
incr_num : num ref → num
decr_num : num ref → num
```

When one has created a reference containing a number (whose type is `num`), it is convenient to increment and decrement its value, using these two (primitive) functions semantically equivalent to:

```
let incr_num i = i := succ_num !i and decr_num i = i := pred_num !i;;
```

7.3.2 Arithmetic operations for nums



```
add_num : num × num → num
minus_num : num → num
sub_num : num × num → num
mult_num : num × num → num
square_num† : num → num
div_num : num × num → num
quo_num : num × num → num
mod_num : num × num → num
```

`add_num`, `minus_num r`, `sub_num` performs respectively the addition, opposite and subtraction on nums.

`mult_num` and `square_num` perform multiplication and square on nums.

`div_num`, `quo_num`, `mod_num` performs respectively the division and Euclidian division quotient and remainder on nums.

Example

```
#add_num (#{1/3}, #{2/3});
#{1/1} : num

#sub_num (#{1/2}, #{1/2});
#{0/1} : num

#square_num #{1/3};
#{1/9} : num

#div_num (#{1/3}, #{1/5});
#{5/3} : num

#quo_num (#{1/3}, #{1/5});
#{1} : num

#mod_num (#{1/3}, #{1/5});
#{2/15} : num
```

7.4 Rounding nums



```
is_integer_num†: num → bool
integer_num†: num → num
floor_num†: num → num
round_num†: num → num
ceiling_num†: num → num
```

`is_integer_num` is equivalent to:

```
let is_integer_num = function
  Ratio r  -> is_integer_ratio r
| _       -> true
;;
```

`integer_num` is the odd function of truncature for rational numbers.

`floor_num` (resp. `round_num`, `ceiling_num`) are the mathematical functions for rounding to the lower (resp. nearest, higher) integer.

Example

```
#is_integer_num #{1.2e3/1200};
true : bool

#integer_num #{1/3};
#{0} : num

#floor_num #{1/3};
#{0} : num

#round_num #{1/3};
#{0} : num
```

```

#ceiling_num #{1/3};;
#{1} : num

#integer_num #{1/2};;
#{0} : num

#floor_num #{1/2};;
#{0} : num

#round_num #{1/2};;
#{1} : num

#ceiling_num #{1/2};;
#{1} : num

#integer_num #{-1/3};;
#{0} : num

#floor_num #{-1/3};;
#{-1} : num

#round_num #{-1/3};;
#{0} : num

#ceiling_num #{-1/3};;
#{0} : num

#integer_num #{-1/2};;
#{0} : num

#floor_num #{-1/2};;
#{-1} : num

#round_num #{-1/2};;
#{-1} : num

#ceiling_num #{-1/2};;
#{0} : num

```

7.5 Comparisons between nums



```

eq_num : num × num → bool
compare_num† : num × num → int
lt_num : num × num → bool
le_num : num × num → bool
gt_num : num × num → bool
ge_num : num × num → bool
sign_num† : num → int
abs_num : num → num

```

min_num : num → num → num

max_num : num → num → num

lt_num, le_num, gt_num, ge_num are arithmetic comparisons corresponding respectively to <, <=, >, >= between elements of type num.

The primitive function for these functions is compare_num, that is semantically equivalent to:

```
let compare_num (r1, r2) =
  when (r1 = r2) -> #0
    | (r1 < r2) -> #-1
    | _ -> #1
;;
```

sign_num is equivalent to:

```
let sign_num = function
  Int i -> sign_int i
| Big_int bi -> sign_big_int bi
| Ratio r -> sign_ratio r
;;
```

abs_num n is the absolute value of n.

max_num (n1, n2) and min_num (n1, n2) are respectively the greater one and the smaller one of rational numbers n1 and n2.

Example

```
#sign_num (succ_num (num_of_int biggest_int));;
```

```
1 : int
```

```
#sign_num (num_of_int monster_int);;
```

```
-1 : int
```

```
#abs_num (succ_num (num_of_int biggest_int));;
```

```
{32768} : num
```

```
#abs_num (num_of_int monster_int);;
```

```
{32768} : num
```

7.6 Coercion functions on nums

7.6.1 Coercion between ints and nums



int_of_num : num → int

num_of_int : int → num

is_int : num → bool

int_of_num n is the int representing the rational number n if n fits in an int.

num_of_int maps an int onto the corresponding num.

is_int is equivalent to:

```
let is_int = function
  Int _ -> true
|   _ -> false
;;
```

Beware:

→ No normalization is performed in this function, so if applied to an `int` with not optimized constructor, the result is `false`. This is possible only if you type in a “small” integer with a label `Big_int` or `Ratio`.

Example

```
#int_of_num #{1.2e3};;
1200 : int

#int_of_num (succ_num (num_of_int biggest_int));;

Evaluation Failed: failure "int_of_big_int"

#int_of_num (pred_num (num_of_int least_int));;
-32768 : int

#num_of_int #1;;
#{1} : num
```

7.6.2 Coercion between floating point numbers and nums

```
float_of_num : num → float
num_of_float : float → num
```

`float_of_num` and `num_of_float` map elements of type `num` into elements of type `float`.

Example

```
#float_of_num #{1.23456789e8};;
123456800.0 : float

#num_of_float it;;
#{123456800} : num
```

7.6.3 Coercion between nats and nums

```
nat_of_num† : num → nat
num_of_nat† : nat → num
```

`nat_of_num n` maps rational number `n` onto the corresponding `nat` if `n` is an integer and positive.
`num_of_nat` maps a `nat` onto the corresponding rational number.

Example

```
#nat_of_num #{1.2e3};;
#<1200> : nat
```

```
#nat_of_num #{-1.2e3};;
```

```
Evaluation Failed: failure "nat_of_int"
```

```
#nat_of_num #{1.2};;
```

```
Evaluation Failed: failure "nat_of_ratio"
```

```
#nat_of_num #{1/2};;
```

```
Evaluation Failed: failure "nat_of_ratio"
```

7.6.4 Coercion between big_ints and nums



big_int_of_num[†]: num → big_int
num_of_big_int[†]: big_int → num

big_int_of_num *n* maps rational number *n* onto the corresponding **big_int** if *n* is an integer else fail.

num_of_big_int maps a integer of type **big_int** onto the corresponding rational number. If this “big” integer fits in a “small” one, this representation is adopted.

Example

```
#big_int_of_num #{1.2};;
```

```
Evaluation Failed: failure "big_int_of_ratio"
```

7.6.5 Coercion between ratios and nums



ratio_of_num[†]: num → ratio
num_of_ratio[†]: ratio → num

ratio_of_num and **num_of_ratio** map a rational number onto its optimized representation.

Example

```
#num_of_ratio #[1];;
#{1} : num
```

```
#ratio_of_num it;;
#[1/1] : ratio
```

7.6.6 Coercion between strings and nums

Reading nums



```

sys_num_of_string†: int × string × int × int → num
num_of_string : string → num

```

CAML is able to read numbers of type `num` entered directly, according to the following lexical convention:

```

Separator ::= ' ' | '\f' | '\n' | '\r' | '\t' | '\\'
Pseudodigit ::= Digit | Separator
Decimal ::= {'-' | '+'} Separator* Digit Pseudodigit*
           {'.' Separator* Digit Pseudodigit*} {'e' {'-' | '+'} INT}
NUM ::= '#{' Decimal {'/' Decimal} '}'

```

Example

```

##{12};;
#{12} : num

##{12.0};;
#{12} : num

##{1.2e1};;
#{12} : num

##{120e-1};;
#{12} : num

##{12/1};;
#{12} : num

##{60/5};;
#{12} : num

##{1.2/1e-1};;
#{12} : num

##{120/10};;
#{12} : num

```

Using non standard arithmetic, the `#{` and `}` characters are optional (see chapter 8 for more information).

`sys_num_of_string (base, s, off, len)` maps the substring `soff,len` in base `base` onto a element of type `num`.

`num_of_string s` is equivalent to:

```

let num_of_string s =
  sys_num_of_string (#10, s, #0, length_string s)
;;

```

Example

```
#sys_num_of_string (#16, "FF", #0, #2);;
#{255} : num

#num_of_string "1.2e3/5e1";;
#{24} : num
```

Printing nums



```
sys_string_of_num†: int × string × num × string → string
string_of_num : num → string
string_for_read_of_num : num → string
display_num : num → unit
echo_num : num → unit
sys_print_num†: int × string × num × string → unit
print_num : num → unit
print_num_for_read : num → unit
```

`sys_string_of_num (base, before, r, after)` writes into the resulting string first the string `before`, then the representation of rational number `n` in base `base`, and last the string `after`. In fact if the approximation when printing rational numbers is turned on, the result is the floating point approximation to the floating point precision of the rational number. This feature is not implemented with `base ≠ 10`. For more information see section 6.9 and chapter 8.

`string_of_num n` is semantically equivalent to:

```
let string_of_num n =
  sys_string_of_num (#10, "", n, "")
;;
```

`string_for_read_of_num n` is analogous to `string_of_num n` except that the number is written between `#{` and `}`, according to conventions to write “big” numbers.

It is equivalent to:

```
let string_for_read_of_num r =
  sys_string_of_num (#10, "#{", r, "}")
;;
```

`display_num` and `echo_num` are equivalent to:

```
let display_num n = display_string (string_of_num n)

and echo_num n = echo_string (string_of_num n)
;;
```

For more information on the differences between `display`, `echo`, `print`-* like functions see the corresponding chapter of the reference manual.

`sys_print_of_num (base, before, r, after)` prints the `base`-radix representation of the rational number `n` between strings `before` and `after`. It is equivalent to:

```
let sys_print_num (base, before, r, after) =
  print_string (sys_string_of_num (base, before, r, after))
;;
```

`print_num` and `print_num_for_read` are equivalent to:

```
let print_num n = sys_print_num (#10, "", n, "");;
let print_num_for_read n = sys_print_num (#10, "#{", n, "}");;
```

Example

```
#sys_string_of_num
# (#16, "num 16-radix representation = ", #{1.2e3/1.1e2}, "");;
"num 16-radix representation = 78/B" : string

#sys_string_of_num
# (#2, "num 2-radix representation = ", #{1.2e3/1.1e2}, "");;
"num 2-radix representation = 1111000/1011" : string

#sys_string_of_num
# (#8, "num 8-radix representation = ", #{1.2e3/1.1e2}, "");;
"num 8-radix representation = 170/13" : string

#sys_string_of_num
# (#10, "num 10-radix representation = ", #{1.2e3/1.1e2}, "");;
"num 10-radix representation = 120/11" : string

#string_of_num #{1.2e3/1.1e2};;
"120/11" : string

#string_for_read_of_num #{1.2e3/1.1e2};;
"#{120/11}" : string

#sys_print_num (#16, "num 16-radix representation = ", #{1.2e3/1.1e2}, "");;
num 16-radix representation = 78/B() : unit

#print_num #{1.2e3/1.1e2};;
120/11() : unit

#print_num_for_read #{1.2e3/1.1e2};;
#{120/11}() : unit
```

Special printing functions



```
sys_latex_print_num†: int × string × num × string → unit
latex_print_num†: num → unit
latex_print_for_read_num†: num → unit
sys_print_beautiful_num†: int × string × num × string → unit
print_beautiful_num†: num → unit
```

These special printing functions for `nums` can be found into two libraries:

`format_latex_numbers.ml`: functions that split numbers into small lines so \LaTeX is able to handle them correctly,

`format_numbers.ml`: functions to present the rational number as a numerical table according to common conventions.

`sys_latex_print_num` (`base`, `before`, `r`, `after`) prints `base` representation of the rational number `n` between `before` and `after`, cutting line each `latex_margin` characters.

`latex_print_num` and `latex_print_for_read_num` are equivalent to:

```
let latex_print_num r = sys_latex_print_num (#10, "", r, "")

and latex_print_for_read_num r = sys_latex_print_num (#10, "#{", r, "}")
;;
```

As an example this manual uses `latex_print_for_read_num` as the printer for rational numbers.

`sys_print_beautiful_num` (`base`, `before`, `r`, `after`) prints `base`-radix representation of the rational number `n` between `before` and `after` as a numerical table according to classical conventions.

`print_beautiful_num` is equivalent to:

```
let print_beautiful_num r =
  sys_print_beautiful_num (#10, "", r, "")
;;
```

You can load the libraries with:

```
load_lib_file "format_latex_numbers";;
load_lib_file "format_numbers";;
```

Example

```
#print_beautiful_num #{1e562};;
10000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000

00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000

00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 000
() : unit

#print_beautiful_num #{1.2e3/1.1e2};;
120/11() : unit

#print_beautiful_num #{12345e105/1.1e2};;
12345 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
```

```

00000 0000
/11() : unit

#print_beautiful_num #{12345e1/1.1e201};;
2469/
22000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
() : unit

```

7.7 Miscellaneous power functions



power_num_int[†]: num × int → num

power_num_big_int[†]: num × big_int → num

power_num : num × num → num

`power_num_int (n, i)` and `power_num_big_int (n, bi)` yield the rational number resulting respectively from the computation of n^i and n^{bi} .

`power_num (n1, n2)` yields the rational number resulting from the computation of $n1^{n2}$ where `n2` is an integer, possibly a “big” one.

Example

```

#power_num_int (#{1.09},#10);;
#{236736367459211723401/10000000000000000000} : num

```

```

#power_num_big_int (#{2}, #(32));;
#{0} : num

```

```

#power_num_big_int (#{2}, #(-32));;
#{1/0} : num

```

```

#power_num (#{2/3},#{3});;
#{8/27} : num

```

7.8 Floating point approximations of nums



approx_num_fix : int × num → string

approx_num_exp : int × num → string

float_num[†]: num → num

`approx_num_fix (n, r)` approaches the rational number `n` with a `n` 10-digits fix point representation. This is an odd function and the last digit is round off. In an informal manner, the printing format is:

```
integer_part . decimal_part_with_n_digits
```

`approx_num_exp (n, r)` approaches the rational number `n` with a `n` 10-digits floating point representation. This is an odd function and the last digit is round off. In an informal manner, the printing format is:

```
(+|-) (0. n_first_digits e msd) | (1. n_zeros e (msd+1))
```

`float_num` is equivalent to:

```
let float_num n = num_of_float (float_of_num n)
;;
```

Example

```
#approx_num_fix (#12, #{1.2e3/1.1e2});;
"+10.9090909091" : string
```

```
#approx_num_fix (#-1, #{1.2e3/1.1e2});;
"+1" : string
```

```
#approx_num_fix (#10, #{355/113});;
"+3.1415929204" : string
```

```
#approx_num_exp (#12, #{1.2e3/1.1e2});;
"+0.1090909091e2" : string
```

```
#approx_num_exp (#2, #{1.2e3/1.1e2});;
"+0.11e2" : string
```

```
#approx_num_exp (#10, #{355/113});;
"+0.3141592920e1" : string
```

```
#float_num #{1.2e3/1.1e2};;
#{1090909/100000} : num
```


Chapitre 8

Numerical directives

8.1 Standard arithmetic

`#standard arith ...`

There are 2 arithmetical modes in CAML, identified by the value of the directive `#standard arith`:

- using the standard arithmetic, only small integers and floating point numbers are recognized by default, floating point arithmetic is an approximate arithmetic and small integer arithmetic is an exact ring arithmetic,
- using the non-standard arithmetic, all numbers are recognized by default of type `num`. Rational operations are performed exactly. Transcendental operations are not defined on type `num`: they can only be applied on floating point representations of numbers, using explicit coercions.

Of course, any number can be read as an entity of (almost) any type, using the lexical convention associated to this type (no lexical convention for type `nat`).

By default, the mode is the standard one, but this can be changed using the `#standard arith` directive.

```
#standard arith false;;
```

tells the compiler to use mathematical arithmetic instead of the standard one. Of course,

```
#standard arith true;;
```

undoes this.

For more information, see chapter “Directives and pragmas” in the reference manual.

8.2 Fast arithmetic

```
#fast arith ...
```

The `#fast arith true` directive tells the compiler that from now on, it has to omit tests as far as possible. When accessing vectors or strings tests for range are ellipsed.

By default this flag is set to false.

8.3 Open arithmetic

`#open arith`

This directive provides more arithmetical functions. These additional functions are indicated in this manual by a † after the name of the function.

Its action is irreversible so it is not the arithmetical mode by default and calling this directive several times adds no more functions. This manual is printed in this arithmetic mode.

This directive is called by:

```
#open arith;;
```

8.4 Debugging nat arithmetic

`#arith cautious`

This directive transforms the arithmetic primitives on nats into cautious primitives. In this mode, every arithmetic function on nats verifies the correction of its arguments according to the conditions indicated in chapter 4.

Its action is irreversible so these verifications are not performed by default and calling this directive several times adds no more verifications.

8.5 Arithmetic overloading

```
overload ... with ...
```

```
#arith ...
```

```
-
```



```
get_arith_overloading : unit → arith list
```

```
prefix + : int × int → int
```

```
prefix * : int × int → int
```

```
prefix ** : int × int → int
```

```
prefix - : int × int → int
```

```
prefix / : int × int → int
```

```
prefix <= : int × int → bool
```

```
prefix < : int × int → bool
```

```
prefix >= : int × int → bool
```

```
prefix > : int × int → bool
```

```
float : int → float
```

```
pred : int → int
```

```
succ : int → int
```

```
incr : int ref → int
```

```
decr : int ref → int
```

```
prefix quo : int × int → int
```

```
prefix mod : int × int → int
```

```
max : int → int → int
```

```
min : int → int → int
```

```
abs : int → int
```

prefix power : `int × int → int`

Most of the operations on numbers are applicable to almost every kind of numbers, e.g. basic arithmetic operations like `+`, `*`, `-` and `/`. Some others are specialized for one precise kind of numbers as is the primitive operation `succ_int : int -> int`.

Roughly speaking, the overloading lets the user not worry about all these subtleties about numbers, especially in the non-standard arithmetic mode. On numerical overloading see section 8.5.

Symbols can be overloaded with the `overload` construction as in:

```
overload f with g;;
```

where f is the overloaded symbol and g is the function which is called when $g(x)$ is well-typed in the context of the application of f to argument x .

By default no numerical overloading is performed. The preceding functions are defined on type `int` in standard arithmetic, else `num`.

`#arith arithmetic_type` loads a set of predefined usual overloading declarations for type `arithmetic_type`. The possible types are: `int`, `float`, `big_int`, `ratio`, `num`.

`#arith overloading` loads a set of predefined usual overloading declarations for types `int`, `num` and `float`.

`get_arith_overloading ()` indicates for which numerical types the predefined overloading mechanism is active.

8.6 Normalization of rational numbers during computation



```
get_normalize_ratio†: unit → bool
set_normalize_ratio†: bool → bool
```

`get_normalize_ratio` (resp. `set_normalize_ratio`) consults (resp. modifies) the value of the flag which controls the automatic normalization of rational numbers during computation.

By default, this flag is set to false. This choice will be justified in appendix B.

8.7 Normalization of rational numbers when printing



```
get_normalize_ratio_when_printing†: unit → bool
set_normalize_ratio_when_printing†: bool → bool
```

`get_normalize_ratio_when_printing` (resp. `set_normalize_ratio_when_printing`) consults (resp. modifies) the status of normalization of rational numbers process during printing.

By default, this flag is set to true.

8.8 Floating point approximation when printing rational numbers



```

get_approx_printing : unit → bool
set_approx_printing : bool → bool
get_floating_precision : unit → int
set_floating_precision : int → int

```

`get_approx_printing` (resp. `set_approx_printing`) consults (resp. modifies) the flag of rational printing mode. If turned on, each rational number will be printed as its floating point approximation according to the current default floating point precision flag, consultable (resp. modifiable) with `get_floating_precision` (resp. `set_floating_precision`).

By default, this flag is set to false.

Example

```

#get_approx_printing ();;
false : bool

#set_approx_printing true;;
true : bool

##[2/3];;
#[2/3] : ratio

#get_floating_precision ();;
12 : int

#set_floating_precision #2;;
2 : int

##{1/4};;
#{1/4} : num

```

8.9 Error when a rational denominator is null



```

get_error_when_null_denominator : unit → bool
set_error_when_null_denominator : bool → bool

```

`get_error_when_null_denominator` (resp. `set_error_when_null_denominator`) consults (resp. modifies) the flag which indicates if null denominators causes run-time errors or not.

By default, this flag is set to true.

8.10 Arith_status or how to remember (or use) all this



```

arith_status : unit → unit

```

`arith_status ()` prints the current status of each arithmetic flag and recall how to manage it.

Example

```

#arith_status();;

```

Standard arithmetic --> ON
(returned by `arith_is_standard ()`)
(modifiable with `#standard arith <your choice>`)

Fast arithmetic --> OFF
(returned by `arith_is_fast ()`)
(modifiable with `#fast arith <your choice>`)

Open arithmetic --> OFF
(returned by `arith_is_open ()`)
(modifiable with `#open arith`)

Cautious arithmetic --> OFF
(returned by `arith_is_cautious ()`)
(modifiable with `#arith cautious`)

Arithmetic overloading --> no overloading on numbers
(returned by `get_arith_overloading ()`)
(modifiable with `#arith overloading` or `#arith <the numerical type to be added>`)

Normalization during computation --> OFF
(returned by `get_normalize_ratio ()`)
(modifiable with `set_normalize_ratio <your choice>`)

Normalization when printing --> ON
(returned by `get_normalize_ratio_when_printing ()`)
(modifiable with `set_normalize_ratio_when_printing <your choice>`)

Floating point approximation when printing rational numbers --> ON
(returned by `get_approx_printing ()`)
(modifiable with `set_approx_printing <your choice>`)

Default precision = 2
(returned by `get_floating_precision ()`)
(modifiable with `set_floating_precision <your choice>`)

Error when a rational denominator is null --> ON
(returned by `get_error_when_null_denominator ()`)
(modifiable with `set_error_when_null_denominator <your choice>`)

`()` : unit

Annexe A

Numerical computations using CAML: π as a case-study

A.1 Introduction

Here is one of Ramanujan's formulas for π , one of the fastest known formulas:

$$\frac{1}{\pi} = \sum_{n=0}^{\infty} (-1)^n \frac{12(6n)!}{(n!)^3(3n)!} \frac{13591409 + 545140134n}{(640320^3)^{n+\frac{1}{2}}} = \sum_{n=0}^{\infty} u_n = \lim_{n \rightarrow \infty} S_n$$

with

$$S_n = \sum_{i=0}^n u_i.$$

David and Gregory Chudnovsky have calculated over 2 Billion digits with this formula, setting the current record (see [1] for the 1 Billion record).

A.2 First approach

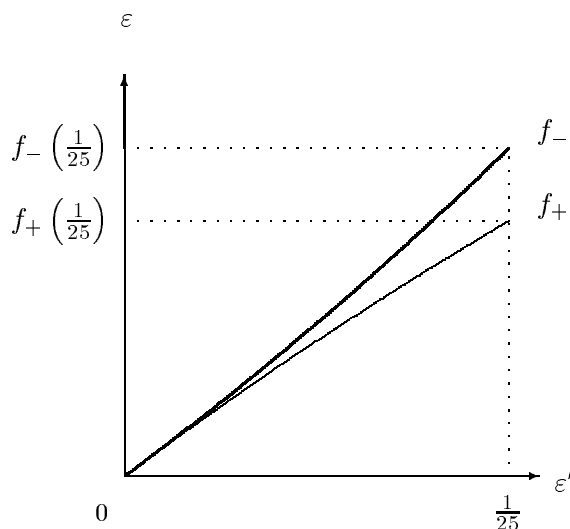
The implementation of this formula raises several problems which are, in increasing order of difficulty:

1. computing $n!$,
2. computing $\sqrt{640320}$ with the necessary precision,
3. determining the number of terms in the series necessary for a given precision.

A.2.1 Factorial function

The first problem is solved with the definition of `fact` as follows:

```
#let rec fact n = if n = 0 then 1 else n * fact (n-1)
#;;
Value fact is <fun> : int -> int
```

Figure A.1 : Graph of $\varepsilon = f(\varepsilon')$.

A.2.2 Square root with a given precision

A square root function is available on numbers of type `big_int`:

```
sqrt_big_int : big_int -> big_int
```

We will use this function to define:

```
#let sqrt640320 digits =
# let pow = 10**digits in
#   num_of_big_int (sqrt_big_int (big_int_of_num (640320*pow*pow)))/pow
#;;
Value sqrt640320 is <fun> : num -> num
```

A.2.3 Required number of terms for a given precision

Theoretical stop

Let ε (resp. ε') be the error when evaluating π (resp. $\frac{1}{\pi}$) via $\frac{1}{S_n}$ (resp. S_n). Let us try to bound ε' above by a simple function of ε . We have:

$$\varepsilon = \left| \pi - \frac{1}{S_n} \right| = \frac{\pi}{|S_n|} \left| S_n - \frac{1}{\pi} \right| = \frac{\pi \varepsilon'}{|S_n|} = \begin{cases} \frac{\pi^2 \varepsilon'}{1 + \pi \varepsilon'} & \text{if } S_n \geq \frac{1}{\pi} \\ \frac{\pi^2 \varepsilon'}{1 - \pi \varepsilon'} & \text{otherwise} \end{cases}$$

Let f_+ and f_- be each of these functions with the obvious sign convention. Figure A.1 represents the graph of these two functions on the interval $[0, \frac{1}{25}]$.

Let f be either f_+ or f_- according to the related position of S_n and $\frac{1}{\pi}$. We see that f is a strictly increasing and continuous function on the interval $[0, \frac{1}{25}]$, so f is invertible and its reciprocal function f^{-1} is strictly increasing and we have:

$$0 \leq \varepsilon \leq \alpha \iff 0 \leq \varepsilon' \leq f^{-1}(\alpha).$$

The reciprocal functions are the following:

$$\begin{aligned} f_+^{-1}(\alpha) &= \frac{\alpha}{\pi(\pi - \alpha)} \\ f_-^{-1}(\alpha) &= \frac{\alpha}{\pi(\pi + \alpha)}. \end{aligned}$$

For our purpose, it is sufficient to have:

$$f^{-1}(\alpha) \geq \frac{\alpha}{\pi(\pi + \frac{1}{25})} \geq \frac{\alpha}{10} = g^{-1}(\alpha)$$

and the preceding equivalence becomes:

$$0 \leq \varepsilon' \leq g^{-1}(\alpha) \leq f^{-1}(\alpha) \iff 0 \leq \varepsilon \leq f(g^{-1}(\alpha)) \leq \alpha.$$

Hence a sufficient condition on ε' for ε to be smaller than α for example, is:

$$\varepsilon' \leq \frac{\alpha}{10}.$$

So we know when the required precision is obtained since this series converges according to the alternating series criterion :

$$\varepsilon' = \left| \frac{1}{\pi} - S_n \right| < |u_{n+1}|.$$

And quite finally,

$$|u_{n+1}| \leq \frac{\alpha}{10} \implies \left| \pi - \frac{1}{S_n} \right| < \alpha$$

and we need to compute $\frac{1}{\pi}$ to a precision of one more digit than the required precision on π .

Practical stop

In fact, the computed term `termn` is different of u_n , since

$$\mathbf{term}_n = \frac{a_n}{\sqrt{640320 - \varepsilon''}},$$

with $\varepsilon'' > 0$ since `sqrt_big_int` (so `sqrt640320`) approximate the integer square root by the integer below it.

Hence, $|\mathbf{term}_n|$ is greater than $|u_n|$ and

$$|\mathbf{term}_n| \leq \frac{\alpha}{10} \implies \left| \pi - \frac{1}{S_n} \right| < \alpha.$$

So it is sufficient to test if the computed term `term` is small enough.

A.2.4 The program

```

##standard arith false;;
Pragma false : bool

##open arith;;
Pragma () : unit

#let rec fact n = if n = 0 then 1 else n * fact (n-1)
#;;
Value fact is <fun> : num -> num

#let sqrt640320 digits =
# let pow = 10**digits in
#   num_of_big_int (sqrt_big_int (big_int_of_num (640320*pow*pow)))/pow
#;;
Value sqrt640320 is <fun> : num -> num

>(* The rational approximation *)
#let approx_pi digits =
# let s = ref 0
# and term = ref (12*13591409/(640320 * (sqrt640320 (digits+1))))
# and n = ref 1 in
#   while abs(!term) > 10**-(digits+1) do
#     s := !s + !term;
#     term := (-1)**!n * 12 * fact (6**!n) * (13591409 + 545140134**!n) /
#       ((fact !n)**3 * fact (3**!n) * (640320**(3**!n+1)) *
#         (sqrt640320 (digits+1)));
#     n := !n+1
#   done;
#   1/!s
#;;
Value approx_pi is <fun> : num -> num

>(* For floating point representation with all digits exacts *)
#let show_pi digits =
# approx_num_fix (int_of_num digits, approx_pi digits)
#;;
Value show_pi is <fun> : num -> string

```

A.2.5 Running the program

Let us use this program for calculating 500 digits of π :

```

>(* Set the timer *)
#timers true;;
() : unit

#approx_pi 500;;
#{109533578894262971661206045498256604078365325503362726370262155560667384279186651900779927
  570310190946061842438218939937169222525519183014235012866529280008563672816062819257319798
  971669125103765337473687536585524549496603999150548618506290892909133189481696463521578380

```

```

144872294713300666551468885759002146646457610371515351903907926614354044167660982964372879
240542917192487914192265573674430376626307022979051823154922233837172938660827803237928588
960499838866382318758304273547899275885838705627819380580508530221364444380195143799170702
088767667865774325800956572314153523124498216651237059970774568175763678533791772677153745
405357769547050973018771813939859104775327100414180701049668744127284139814454178213319826
644995908057908345165764473116510815311598331576031991889893973653676085974423277305563664
564116488796814040993607972020766441302977724130414750003722754981888

```

/

```

348656210311361025844113583946271930042626470328710522455171383259045465585547268971573729
674425713020961732656682830884669904177202439885182857021600872877923638564342437539455025
846478552183120889956595464640351147028471604408180922910361711679446230664637935030803718
995215530925783895047193663963462239373598701691525332442020213154858030496049829381069484
712645395465629267115052143734839709025306060035261044354059221522133302615797035453487543
742256311458194541434367772226200345428239347618367709412811745637085904186881457969045109
760848870336570798472120415374147869990916950073723705791897984250467880654043393275222624
444764342600399176629045742140925661058493518343716381215736402047252333851495752809609896
097235367589044916690716546457687197389910305791784880944555282097991005087635542570267939
174906121649837703323993802310976075631288040312938392162322998046875} :

```

num

Evaluation has needed: Runtime: 17.34s GC: 2.90s

Now let us try with 1000 digits:

```
#approx_pi 1000;;
```

Evaluation Failed: failure "create_nat: number too long"

Evaluation has needed: Runtime: 95.96s GC: 10.35s

Two remarks:

- the running time is long and the computation of 1000 digits fails because the rational numbers used have too big numerator and denominator,
- this program is very redundant; some calculations are done several times.

It's time to optimize this algorithm a bit.

A.3 Second approach

A.3.1 The algorithm

In this section, we will factorize as far as possible the calculation using Horner techniques. We can deduce a factorial from the preceding iteration with less computation and optimize the computation of the power in the denominator.

Let us introduce some notations that will be used in the program too:

$$\begin{aligned}
 \text{sign}_n &= (-1)^n \\
 \text{lin}_n &= 13591409 + 545140134n \\
 \text{six_n_fact}_n &= (6n)! \\
 \text{three_n_fact}_n &= (3n)!
 \end{aligned}$$

$$\begin{aligned}\text{fact_n_pow3}_n &= (n!)^3 \\ \text{pow_n}_n &= 640320^{3n+1}\end{aligned}$$

We will use the following transformation:

$$\begin{aligned}\text{sign}_{n+1} &= -\text{sign}_n \\ \text{lin}_{n+1} &= 545140134 + \text{lin}_n \\ \text{six_n_fact}_{n+1} &= (6n+6)(6n+5)(6n+4)(6n+3)(6n+2)(6n+1) \text{six_n_fact}_n \\ \text{three_n_fact}_{n+1} &= (3n+3)(3n+2)(3n+1) \text{three_n_fact}_n \\ \text{fact_n_pow3}_{n+1} &= (n+1)^3 \text{fact_n_pow3}_n \\ \text{pow_n}_{n+1} &= (640320^3) \text{pow_n}_n\end{aligned}$$

with :

$$\begin{aligned}\text{sign}_0 &= 1 \\ \text{lin}_0 &= 13591409 \\ \text{six_n_fact}_0 &= 1 \\ \text{three_n_fact}_0 &= 1 \\ \text{fact_n_pow3}_0 &= 1 \\ \text{pow_n}_0 &= 640320\end{aligned}$$

and we compute only once the square root and the tenth power for the test.

A.3.2 Stop

With S_n the partial sum of the series without the square root and R_n the approximation of this square root, we try to approach $\pi = \frac{\sqrt{640320}}{S_\infty}$ with $\frac{R_n}{S_n}$. We have :

$$\begin{aligned}S_n &= S_\infty - \varepsilon' \\ R_n &= \sqrt{640320} - \varepsilon'',\end{aligned}$$

where ε'' is a positive number and ε' either positive or negative.

Let us try to bound $|\varepsilon'|$ and ε'' above by a simple function of ε , the error when evaluating π . We have:

$$\pi - \frac{R_n}{S_n} = \frac{\sqrt{640320}}{S_n + \varepsilon'} - \frac{\sqrt{640320} - \varepsilon''}{S_n} = \frac{-\sqrt{640320} \varepsilon' + S_n \varepsilon'' + \varepsilon' \varepsilon''}{S_n(S_n + \varepsilon')}.$$

If we simplify this expression eliminating the references to S_n , we obtain

$$\varepsilon = \left| \pi - \frac{R_n}{S_n} \right| = \left| \frac{-\pi \varepsilon' + \varepsilon''}{\frac{\sqrt{640320}}{\pi} - \varepsilon'} \right| \leq \frac{\varepsilon'' + \pi |\varepsilon'|}{\frac{\sqrt{640320}}{\pi} - |\varepsilon'|} = f_{\varepsilon''}(|\varepsilon'|).$$

Thus

$$f_{\varepsilon''}(|\varepsilon'|) \leq \alpha \implies \varepsilon \leq \alpha.$$

$f_{\varepsilon''}$ is a continuous and strictly increasing function so we have :

$$|\varepsilon'| \leq f_{\varepsilon''}^{-1}(\alpha) \implies f_{\varepsilon''}(|\varepsilon'|) \leq \alpha.$$

The reciprocal function $f_{\varepsilon''}^{-1}$ is

$$f_{\varepsilon''}^{-1}(\alpha) = \frac{\frac{\sqrt{640320}}{\pi} \alpha - \varepsilon''}{\pi + \alpha}.$$

Under the condition

$$\varepsilon'' \leq 100 \alpha,$$

and $\alpha \leq 1$, we have

$$f_{\varepsilon''}^{-1}(\alpha) \geq \frac{\frac{\sqrt{640320}}{\pi} - 100}{\pi + 1} \alpha > 10 \alpha.$$

Hence a sufficient condition on ε'' and $|\varepsilon'|$ for ε to be smaller than α for example, is:

$$\begin{aligned} \varepsilon'' &\leq 100 \alpha \\ |\varepsilon'| &\leq 10 \alpha. \end{aligned}$$

In terms of digits, we need to compute the partial sum and the square root to a precision of respectively one and two decimal digits less than the required precision on π .

A.3.3 The program

```
#let sqrt640320 digits =
# let pow = 10**digits in
#   (num_of_big_int (sqrt_big_int (big_int_of_num (640320*pow*pow))), pow)
#;;
Value sqrt640320 is <fun> : num -> num * num

#let approx_pi digits =
# let s = ref 0
# and sign = ref 1
# and six_n_fact = ref 1
# and three_n_fact = ref 1
# and fact_n_pow3 = ref 1
# and lin = ref (13591409)
# and pow3 = 640320**3 in
# let pow_n = ref 640320
# and term = ref (12*13591409/640320)
# and (sqrt, pow) = (sqrt640320 (digits-2))
# and n = ref 0 in
#   let pow_test = pow*10 in
#   while abs(!term)*pow_test > 1 do
#     s := !s + !term;
#     lin := 545140134 + !lin;
#     sign := - !sign;
#     six_n_fact := (6*!n+6)*(6*!n+5)*(6*!n+4)*(6*!n+3)*(6*!n+2)*(6*!n+1)
#                   *!six_n_fact;
#     three_n_fact := (3*!n+3)*(3*!n+2)*(3*!n+1)*!three_n_fact;
```

```

#      pow_n := pow3!*pow_n;
#      term := (12 * !sign * !six_n_fact * !lin) /
#              (!fact_n_pow3 * !three_n_fact * !pow_n);
#      fact_n_pow3 := (!n+1)**3*!fact_n_pow3;
#      n := !n+1
#      done;
#      sqrt(!s*pow)
#;;
Value approx_pi is <fun> : num -> num

```

A.3.4 Running the program

```

#echo_values false;;
: unit

```

```

#approx_pi 500;;
: num
Evaluation has needed: Runtime: 5.53s GC: 1.91s

```

```

#approx_pi 1000;;
: num
Evaluation has needed: Runtime: 61.11s GC: 7.64s

```

Now let us try with 2000 digits:

```

#approx_pi 2000;;

Evaluation Failed: failure "create_nat: number too long"
Evaluation has needed: Runtime: 82.65s GC: 13.12s

```

Computing 500 digits of π is now faster, computing 1000 digits does not create too long numbers anymore but computing 2000 digits still create too long integers. So we must improve the algorithm once again.

A.4 Third approach

A.4.1 The algorithm

We remark that:

$$\begin{aligned}
 (n+1)^3 &= (3n^2 + 3n + 1) + n^3 \\
 3(n+1)^2 + 3(n+1) + 1 &= (6n+6) + (3n^2 + 3n + 1) \\
 6(n+1) + 6 &= 6 + (6n+6)
 \end{aligned}$$

Let T_n , pown3_n , binom_n and sixn_n be the following quantities:

$$\begin{aligned}
 T_n &= \frac{(-1)^n 12 (6n)!}{(n!)^3 (3n)! (640320)^{3n+1}} \\
 \text{lin}_n &= 13591409 + 545140134n
 \end{aligned}$$

$$\begin{aligned} \text{pown3}_n &= n^3 \\ \text{binom}_n &= 3n^2 + 3n + 1 \\ \text{sixn}_n &= 6n \end{aligned}$$

We have :

$$\begin{aligned} T_{n+1} &= \frac{-8 \times (\text{sixn}_n + 5) (\text{sixn}_n + 3) (\text{sixn}_n + 1)}{\text{pown3}_{n+1} \times \text{pow3}} T_n \\ \text{lin}_{n+1} &= 545140134 + \text{lin}_n \\ \text{pown3}_{n+1} &= \text{binom}_n + \text{pown3}_n \\ \text{binom}_{n+1} &= \text{sixn}_{n+1} + \text{binom}_n \\ \text{sixn}_{n+1} &= 6 + \text{sixn}_n \end{aligned}$$

with

$$\begin{aligned} T_0 &= \frac{12}{640320} \\ \text{lin}_0 &= 13591409 \\ \text{pown3}_0 &= 0 \\ \text{binom}_0 &= 1 \\ \text{sixn}_0 &= 0. \end{aligned}$$

A.4.2 The program

```
#let approx_pi digits =
# let s = ref 0
# and sixn = ref 0
# and T = ref (12/640320)
# and binom = ref 1 (* 3n^2+3n+1 *)
# and pown3 = ref 0 (* n^3 *)
# and lin = ref (13591409) (* 13591409+545140134n *)
# and pow3 = 640320**3
# and (sqrt, pow) = sqrt640320 (digits - 2)
# and term = ref (12*13591409/640320) in
# let pow_test = pow*10 in
# while abs(!term)*pow_test > 1 do
#   s := !s + !term;
#   pown3 := !binom + !pown3;
#   T := -8 * (!sixn+1) * (!sixn+3) * (!sixn+5) / (!pown3 * pow3) * !T;
#   lin := 545140134 + !lin;
#   term := !T * !lin;
#   sixn := !sixn + 6;
#   binom := !sixn + !binom
#   done;
# sqrt(!s*pow)
#;;
Value approx_pi : num -> num
```

A.4.3 Running the program

```
#approx_pi 500;;
: num
Evaluation has needed: Runtime: 4.20s GC: 1.91s
```

```
#approx_pi 1000;;
: num
Evaluation has needed: Runtime: 40.59s GC: 6.61s
```

Now let us try it with 2000 digits:

```
#approx_pi 2000;;

Evaluation Failed: failure "create_nat: number too long"
Evaluation has needed: Runtime: 81.30s GC: 15.05s
```

The runtime decreases but there are still too many allocation of numbers. So the next step is computation with no allocation of numbers of type `nat`. We will prepare this step using first signed integers.

A.5 Fourth approach

A.5.1 The algorithm

Let us note that each partial sum is naturally a quotient of two integers N_n and D_n with :

$$\begin{aligned}
 D_n &= 640320^{3n+1} (n!)^3 \\
 \text{prod}_n &= (-1)^n \frac{12 (6n)!}{(3n)!} \\
 \text{sum}_n &= 13591409 + 545140134n \\
 N_n &= 640320^{3n} n!^3 \sum_{i=0}^{i=n} (-1)^i \frac{12(6i)!}{(3i)!} \frac{13591409 + 545140134i}{640320^{3i} i!^3} \\
 \text{pown3}_n &= n^3 \\
 \text{binom}_n &= 3n^2 + 3n + 1 \\
 \text{sixn}_n &= 6n.
 \end{aligned}$$

So we have recursively

$$\begin{aligned}
 D_{n+1} &= 640320^3 (n+1)^3 D_n \\
 \text{prod}_{n+1} &= -8 (6n+1) (6n+3) (6n+5) \text{prod}_n \\
 \text{sum}_{n+1} &= 545140134 + \text{sum}_n \\
 N_{n+1} &= 640320^3 (n+1)^3 N_n + \text{prod}_{n+1} \text{sum}_{n+1} \\
 \text{pown3}_{n+1} &= \text{binom}_n + \text{pown3}_n \\
 \text{binom}_{n+1} &= \text{sixn}_{n+1} + \text{binom}_n \\
 \text{sixn}_{n+1} &= 6 + \text{sixn}_n
 \end{aligned}$$

and

```

D0 = 640320
prod0 = 12
sum0 = 13591409
N0 = 12 × 13591409
pown30 = 0
binom0 = 1
sixn0 = 0.

```

We stop the computation when

$$\left| \frac{\text{prod}_n \text{sum}_n}{D_n} \right| \leq 100 \alpha.$$

We have $\alpha = 10^{-\text{digits}}$ so we test

$$\left| \text{prod}_n \text{sum}_n 10^{\text{digits}-1} \right| \leq D_n.$$

A.5.2 The program

```

#let approx_pi digits =
# let prod = ref 12
# and sum = ref 13591409
# and D = ref 640320
# and N = ref (12*13591409)
# and sixn = ref 0
# and binom = ref 1 (* 3n^2+3n+1 *)
# and pown3 = ref 0 (* n^3 *)
# and (sqrt, pow) = sqrt640320 (digits-2)
# and pow3 = 640320**3 in
# let pow_test = pow*10 in
# while abs(!prod*!sum*pow_test) > !D do
#   prod := -8*(!sixn+1)*(!sixn+3)*(!sixn+5)*!prod;
#   sum := 545140134+!sum;
#   pown3 := !binom + !pown3;
#   D := !pown3*pow3*!D;
#   N := !pown3*pow3*!N+!prod*!sum;
#   sixn := !sixn+6;
#   binom := !sixn + !binom
#   done;
# sqrt*!D/(!N*pow)
#;;
Value approx_pi : num -> num

```

A.5.3 Running the program

```

#approx_pi 500;;
: num
Evaluation has needed: Runtime: 0.17s

```


Now let us try with 10000 digits:

```
#approx_pi 10000;;
: num
Evaluation has needed: Runtime: 260.32s GC: 48.96s
```

Computing 500 digits of π is now quite fast, but computing 10000 digits takes too much time and too much GC. It uses less room in the heap but the amount of GC is still enormous. So we will now translate the preceding algorithm with `big_int` functions as an intermediary step for type `nat`.

A.5.4 big_int version

We increment `sixn` during the computation of T_{n+1} and we integrate the alternate sign in it. We can consider the denominator as

$$D_{n+1} = (640320 (n+1))^3 D_n.$$

Let us remind the definition of `D`, `prod`, `sum`, `N`, `incr_den`, `binom`, `monom` and `sixn` to us.

$$\begin{aligned} D_n &= 640320^{3n+1} (n!)^3 \\ \text{prod}_n &= (-1)^n \frac{12 (6n)!}{(3n)!} \\ \text{sum}_n &= 13591409 + 545140134n \\ N_n &= 640320^{3n} n!^3 \sum_{i=0}^{i=n} (-1)^i \frac{12(6i)!}{(3i)!} \frac{13591409 + 545140134i}{640320^{3i} i!^3} \\ \text{incr_den}_n &= 640320^3 n^3 \\ \text{binom}_n &= 640320^3 (3n^2 + 3n + 1) \\ \text{monom}_n &= 640320^3 (6n) \\ \text{sixn}_n &= 6n + 1. \end{aligned}$$

So we have recursively

$$\begin{aligned} D_{n+1} &= 640320^3 (n+1)^3 D_n \\ \text{prod}_{n+1} &= -8 (6n+1) (6n+3) (6n+5) \text{prod}_n \\ \text{sum}_{n+1} &= 545140134 + \text{sum}_n \\ N_{n+1} &= 640320^3 (n+1)^3 N_n + \text{prod}_{n+1} \text{sum}_{n+1} \\ \text{incr_den}_{n+1} &= \text{binom}_n + \text{incr_den}_n \\ \text{binom}_{n+1} &= \text{monom}_{n+1} + \text{binom}_n \\ \text{monom}_{n+1} &= \text{monom}_n + \text{six_pow3} \\ \text{sixn}_{n+1} &= 6 + \text{sixn}_n \end{aligned}$$

and

$$D_0 = 640320$$

```

prod0 = 12
sum0 = 13591409
N0 = 12 × 13591409
incr_den0 = 0
binom0 = 262537412640768000
monom0 = 0
sixn0 = 1.

```

We simplify the stop test, we determine an upper bound `sizeB` on the size of the 2^{32} -radix representation of this number and then we test if the product `prod × sum` shifted by `sizeB` digits is less than `D`. For our purpose, it is sufficient to have:

$$size(\text{prod}) + size(\text{sum}) + \text{sizeB} < size(D).$$

```

##standard arith true;;
Pragma : bool

##open arith;;
Pragma : unit

#let test (prod, sum, D, sizeB) =
# (num_digits_big_int prod)+(num_digits_big_int sum)+sizeB >
# (num_digits_big_int D)
#;;
Value test : big_int * big_int * big_int * int -> bool

#let sqrt640320 digits =
# let pow = power_int_positive_int (#10, digits) in
#   create_ratio (sqrt_big_int (mult_big_int (#(640320), square_big_int pow)),
#               pow)
#;;
Value sqrt640320 : int -> ratio

#let approx_pi digits =
# let prod = ref #(12)
# and sum = ref #(13591409)
# and sixn = ref #1
# and N = ref #(163096908)
# and D = ref #(640320)
# and incr_den = ref #(0) (* (640320n)^3 *)
# and binom = ref #(262537412640768000) (* incr_den = incr_den + binom *)
# and monom = ref #(0) (* binom = binom + monom *)
# and six_pow3 = #(1575224475844608000) (* monom = monom + six_pow3,
#                                       six_pow3=640320^3 * 6 *)
# and d = max_int #0 (pred_int digits) in
# let sizeB = (* 10^d <= (2^32)^sizeB *)
#   int_of_big_int (ceiling_ratio (mult_int_ratio (d, #[3.322/32]))) in
#   while test (!prod, !sum, !D, sizeB) do
#     prod := mult_int_big_int (#-8, !prod);

```

```

#   prod := mult_int_big_int (!sixn, !prod);
#   sixn := add_int (!sixn, #2);
#   prod := mult_int_big_int (!sixn, !prod);
#   sixn := add_int (!sixn, #2);
#   prod := mult_int_big_int (!sixn, !prod);
#   sixn := add_int (!sixn, #2);
#   sum := add_big_int (!sum, #(545140134));
#   incr_den := add_big_int (!incr_den, !binom);
#   monom := add_big_int (!monom, six_pow3);
#   binom := add_big_int (!binom, !monom);
#   N := add_big_int (mult_big_int (!prod, !sum),
#                     mult_big_int (!incr_den, !N));
#   D := mult_big_int (!incr_den, !D)
# done;
# let r = (sqrt640320 (pred_int d)) in
#   create_ratio (mult_big_int (numerator_ratio r, !D),
#                 mult_big_int (denominator_ratio r, !N))
#;;
Value approx_pi : int -> ratio

```

A.5.5 Running the program

```

#approx_pi #10000;;
: ratio
Evaluation has needed: Runtime: 60.61s GC: 34.18s

```

Now let us try with 20000 digits:

```

#approx_pi #20000;;
: ratio
Evaluation has needed: Runtime: 179.98s GC: 171.16s

```

The runtime is quite better, but the GC-time is almost equal to the runtime for 20000 digits, so we will now improve the GC-time with type `nat`.

A.6 Final version: type `nat`

We need to know an upper bound of the size of the value of each variable. These sizes are increasing functions of the number of iterations so we will now compute an upper bound for this number and for the ultimate size of each variable.

A.6.1 Number of iterations

The series

$$\frac{S_n}{\sqrt{640320}} = \frac{\sum_{n=0}^{\infty} (-1)^n a_n}{\sqrt{640320}} = \sum_{n=0}^{\infty} (-1)^n \frac{12(6n)!}{(n!)^3(3n)!} \frac{13591409 + 545140134n}{640320^{3n + \frac{3}{2}}}$$

converges according to the alternated series criterion so we have:

$$\varepsilon' = \left| \frac{\sqrt{640320}}{\pi} - S_n \right| < a_{n+1},$$

and it is sufficient that an upper bound of a_n is less than the precision needed. We try to find a simple upper bound of a_n .

According to Stirling's formula:

$$\left(\frac{n}{e}\right)^n \sqrt{2\pi n} < n! < \left(\frac{n}{e}\right)^n \sqrt{2\pi n} e^{\frac{1}{12n}}$$

we have

$$\begin{aligned} (6n)! &< \left(\frac{6n}{e}\right)^{6n} \sqrt{12\pi n} e^{\frac{1}{72n}} \\ n!^3 &> \left(\frac{n}{e}\right)^{3n} (\sqrt{2\pi n})^3 \\ (3n)! &> \left(\frac{3n}{e}\right)^{3n} \sqrt{6\pi n} \end{aligned}$$

and

$$\frac{(6n)!}{n!^3(3n)!} < \frac{12^{3n} e^{\frac{1}{72n}}}{2 (\sqrt{\pi n})^3}.$$

So we have

$$a_n < \frac{1}{2\pi^{\frac{3}{2}}} \left(\frac{12}{640320}\right)^{3n+1} \frac{e^{\frac{1}{72n}} (13591409 + 545140134n)}{n^{\frac{3}{2}}}.$$

Hence

$$a_n < \frac{e^{\frac{1}{72}} 558731543}{106720 \pi^{\frac{3}{2}}} \frac{1}{n^{\frac{1}{2}} (151931373056000)^n}$$

if we bound above $(13591409 + 545140134n)$ by $(13591409 + 545140134) \times n$ and $e^{\frac{1}{72n}}$ by $e^{\frac{1}{72}}$. Let λ and x respectively be

$$\lambda = \frac{e^{\frac{1}{72}} 558731543}{106720 \pi^{\frac{3}{2}}}$$

and $x = 151931373056000$, the preceding inequality becomes:

$$a_n < \lambda \frac{1}{n^{\frac{1}{2}} x^n}.$$

So a sufficient condition for $u_n < \alpha$ is

$$n x^{2n} > \frac{\lambda^2}{\alpha^2}$$

and a fortiori

$$x^{2n} > \frac{\lambda^2}{\alpha^2},$$

that is to say

$$n \geq \left\lceil \frac{\ln_B(\lambda) - \ln_B(\alpha)}{\ln_B(x)} \right\rceil.$$

With $B = 2^{32}$ we have

$$n = \lceil 0.21 + 0.68 \times d \rceil$$

where d is the number of precision digits in base B for $\frac{\sqrt{640320}}{\pi}$ so $d = \text{digits} - 2$ where digits is the number of precision digits in base B for π and

$$n = \lceil -1.14 + 0.68 \times \text{digits} \rceil \text{ with } B = 2^{32}.$$

We can notice that the successive upper bounds we have determine were quite good since this final upper bound for n is quasi-optimal.

A.6.2 Size of the objects with base 2^{32}

We consider as a reasonable value of n , any integer smaller than 3000 according to the maximum number of digits. So $\ln_B(n)$ is bounded above by $\ln_B(3000)$ from now on.

Let us remark that 640320^3 and $\frac{(6n)!}{(3n)!}$ ($n \geq 1$) have at least 24 as common factor. If $n = 1$, 24 divides $\frac{(6n)!}{(3n)!} = 6 \times 5 \times 4$, and if $n \geq 2$,

$$\binom{6n}{3n} = \frac{(6n)!}{(3n)! (3n)!}$$

is an integer so

$$\frac{(6n)!}{(3n)!} = (3n)! \binom{6n}{3n}$$

is a multiple of $(3n)!$ so a multiple of $2 \times 3 \times 4 = 24$. So we can reduce the size and the gcd of N (hence prod too) and D if we simplify them by a factor 24 each step.

Recall the definition of D , prod , sum , N , incr_den , binom , monom and sixn to us.

$$\begin{aligned} D_n &= \left(\frac{640320}{12} \right) \times \left(\frac{640320^3}{24} \right)^n (n!)^3 \\ \text{prod}_n &= \frac{(6n)!}{24^n (3n)!} \\ \text{sum}_n &= 13591409 + 545140134n \\ N_n &= 640320^{3n} n!^3 \sum_{i=0}^{i=n} (-1)^i \frac{12(6i)!}{(3i)!} \frac{13591409 + 545140134i}{640320^{3i} i!^3} \\ \text{incr_den}_n &= \frac{640320^3}{24} n^3 \\ \text{binom}_n &= \frac{640320^3}{24} (3n^2 + 3n + 1) \\ \text{monom}_n &= \frac{640320^3}{24} (6n) \\ \text{sixn}_n &= 6n \\ \text{twon}_n &= 2n + 1. \end{aligned}$$

So we have recursively

$$\begin{aligned}
D_{n+1} &= \text{incr_den}_{n+1}; D_n \\
\text{prod}_{n+1} &= (6n+1)(6n+5) \text{twon}_n \text{prod}_n \\
\text{sum}_{n+1} &= 545140134 + \text{sum}_n \\
N_{n+1} &= \text{incr_den}_{n+1} N_n + \text{prod}_{n+1} \text{sum}_{n+1} \quad \text{if } n+1 \text{ is even} \\
&= \text{incr_den}_{n+1} N_n - \text{prod}_{n+1} \text{sum}_{n+1} \quad \text{otherwise} \\
\text{incr_den}_{n+1} &= \text{binom}_n + \text{incr_den}_n \\
\text{binom}_{n+1} &= \text{monom}_{n+1} + \text{binom}_n \\
\text{monom}_{n+1} &= \text{monom}_n + \text{six_pow3} \\
\text{sixn}_{n+1} &= 6 + \text{sixn}_n \\
\text{twon}_{n+1} &= 2 + \text{twon}_n
\end{aligned}$$

and

$$\begin{aligned}
D_0 &= 53360 \\
\text{prod}_0 &= 1 \\
\text{sum}_0 &= 13591409 \\
N_0 &= 13591409 \\
\text{incr_den}_0 &= 0 \\
\text{binom}_0 &= 10939058860032000 \\
\text{monom}_0 &= 0 \\
\text{sixn}_0 &= 0.
\end{aligned}$$

The ultimate value of $\text{sum} = 13591409 + 545140134n$ takes 1 digit if $n \leq 7$ and 2 for all other reasonable values of n .

twon and sixn take only a small `int` for the reasonable values of n , but they are used as a digit.

We have :

$$\text{prod}_n = \frac{(6n)!}{24^n (3n)!} < \sqrt{2} \left(\frac{72n^3}{e} \right)^n e^{\frac{1}{72n}}$$

so

$$\ln_B(\text{prod}_n) < \ln_B \left(\frac{72 \times 3000^3}{e} \right) n + \ln_B \left(\sqrt{2} e^{\frac{1}{72}} \right) < [1.23061n + 0.01625117].$$

We have :

$$D_n = 10939058860032000^n \times 53360 \times n!^3 < 10939058860032000^n \times 53360 \times \left(\frac{n}{e} \right)^{3n} (\sqrt{2\pi n})^3 e^{\frac{1}{4n}}$$

so

$$\ln_B(D_n) < \ln_B \left(\frac{10939058860032000 \times 3000^3}{e^3} \right) n + \ln_B \left(53360 (\sqrt{2 \times 3000 \pi})^3 e^{\frac{1}{4}} \right) < [2.613n + 1.168].$$

monom , binom and incr_den take less than 3 digits.

N takes $\lceil 2.613n + 1.418 \rceil$ digits.

In fact, the final product with the square root is performed in N and D so the reserved size for these variables is increased by the number of digits.

Let us notice that the precision of `digits - 2` for the square root was for $B = 10$ and we need to take `digits` of precision for $B = 2^{32}$.

A.6.3 The complete program

```
##standard arith true;;
Pragma : bool

##fast arith true;;
Pragma : unit

##open arith;;
Pragma : unit

#let sqrt640320 digits =
# let len = mult_int (#2, digits) in
# let real_len = succ_int len in
# let nat = create_nat real_len in
#   set_digit_nat (nat, len, #10005);
#   shift_left_nat (nat, #0, real_len, create_nat #1, #0, #6);
#   (* now nat = 10005*2^6*2^(32*digits) *)
#   sqrt_nat (nat, #0, real_len)
#;;
Value sqrt640320 : int -> nat

#let approx_pi digits =
# (* Number of digits *)
# let iter = int_of_big_int (
#   ceiling_ratio (
#     add_ratio (#[-1.14],
#       mult_int_ratio (digits, #[0.68]))) in
# (* Sizes *)
# let size_sum = if le_int (iter, #7) then #1 else #2
# and max_size_prod = int_of_big_int (
#   ceiling_ratio (
#     add_ratio (#[0.01625117],
#       mult_int_ratio (iter, #[1.23061])))
# and common = mult_int_ratio (iter, #[2.613]) in
# let max_sizeD = succ_int (add_int (digits, int_of_big_int (
#   ceiling_ratio (add_ratio (#[1.168], common))))
# and max_sizeN = add_int (digits, int_of_big_int (
#   ceiling_ratio (add_ratio (#[1.418], common)))) in
# (* Creations *)
# let prod = make_nat max_size_prod
# and sum = make_nat size_sum
# and N = ref (make_nat max_sizeN)
# and D = ref (make_nat max_sizeD)
```

```

# and incr_den = make_nat #3
# and binom = make_nat #3
# and monom = make_nat #3
# and six_pow3 = nat_of_string "65634353160192000"
# and sixn = make_nat #1
# and twon = make_nat #1
# and incr_sum = nat_of_string "545140134"
# and aux1 = ref (make_nat max_sizeN) (* aux1 and aux2 are auxiliary *)
# and aux2 = ref (make_nat max_sizeN) (* variables for computing N and D *)
# and size_aux1 = ref #1
# and size_aux2 = ref #1
# and size_prod = ref #1
# and new_size_prod = ref #1
# and sizeN = ref #1
# and sizeD = ref #1
# in
#   (* Initialisation *)
#   set_digit_nat (prod, #0, #1);
#   blit_nat (sum, #0, nat_of_string "13591409", #0, #1);
#   blit_nat (!N, #0, nat_of_string "13591409", #0, #1);
#   blit_nat (!D, #0, nat_of_string "53360", #0, #1);
#   blit_nat (incr_den, #0, nat_of_string "10939058860032000", #0, #2);
#   blit_nat (binom, #0, nat_of_string "10939058860032000", #0, #2);
#   (* Treatment *)
#   for i = 1 to iter do
#     (* Treatment of prod *)
#     new_size_prod := min_int max_size_prod (succ_int !size_prod);
#     mult_digit_nat (prod, #0, !new_size_prod,
#                   prod, #0, !new_size_prod, sixn, #0);
#     incr_nat (sixn, #0, #1, #1);
#     incr_nat (sixn, #0, #1, #1);
#     incr_nat (sixn, #0, #1, #1);
#     incr_nat (sixn, #0, #1, #1);
#     mult_digit_nat (prod, #0, !new_size_prod,
#                   prod, #0, !new_size_prod, sixn, #0);
#     incr_nat (sixn, #0, #1, #1);
#     incr_nat (sixn, #0, #1, #1);
#     mult_digit_nat (prod, #0,
#                   (min_int max_size_prod (succ_int !new_size_prod)),
#                   prod, #0, !new_size_prod, twon, #0);
#     incr_nat (twon, #0, #1, #1);
#     incr_nat (twon, #0, #1, #1);
#     size_prod :=
#       num_digits_nat (prod, #0,
#                     (min_int max_size_prod (succ_int !new_size_prod)));
#     (* Treatment of sum *)
#     add_nat (sum, #0, size_sum, incr_sum, #0, #1, #0);
#     (* Treatment of N *)
#     mult_nat (!aux1, #0, add_int (!size_prod, size_sum),
#             prod, #0, !size_prod, sum, #0, size_sum);
#     size_aux1 := num_digits_nat (!aux1, #0, add_int (!size_prod, size_sum));

```



```

#      (exchange N aux2);
#      (exchange sizeN size_aux2);
#      mult_nat (!N, #0, add_int (!size_aux2, #3),
#              !aux2, #0, !size_aux2, incr_den, #0, #3);
#      sizeN := num_digits_nat (!N, #0, add_int (!size_aux2, #3));
#      (if (i mod 2 = 0)
#          then add_nat (!N, #0, succ_int (max_int !sizeN !size_aux1),
#                      !aux1, #0, !size_aux1, #0)
#          else sub_nat (!N, #0, !sizeN, !aux1, #0, !size_aux1, #1));
#      sizeN := num_digits_nat (!N, #0, succ_int (max_int !sizeN !size_aux1));
#      (* Treatment of D *)
#      set_to_zero_nat (!aux1, #0, !size_aux1);
#      size_aux1 := 1;
#      set_to_zero_nat (!aux2, #0, !size_aux2);
#      size_aux2 := 1;
#      (exchange D aux2);
#      (exchange sizeD size_aux2);
#      mult_nat (!D, #0, add_int (!size_aux2, #3),
#              !aux2, #0, !size_aux2, incr_den, #0, #3);
#      sizeD := num_digits_nat (!D, #0, add_int (!size_aux2, #3));
#      set_to_zero_nat (!aux2, #0, !size_aux2);
#      size_aux2 := 1;
#      (* Treatment of incr_den *)
#      add_nat (monom, #0, #3, six_pow3, #0, #2, #0);
#      add_nat (binom, #0, #3, monom, #0, #3, #0);
#      add_nat (incr_den, #0, #3, binom, #0, #3, #0)
#  done;
#  let sq = sqrt640320 digits in
#    create_ratio (
#      ((exchange D aux2);
#       (exchange sizeD size_aux2);
#       mult_nat (!D, #0, min_int (add_int (!size_aux2, (succ_int digits)))
#                               max_sizeD,
#                               !aux2, #0, !size_aux2, sq, #0, succ_int digits);
#       big_int_of_nat !D),
#      (blit_nat (!N, digits, !N, #0, !sizeN);
#       big_int_of_nat !N))
#  ;;
Value approx_pi : int -> ratio

#let num_of_digits digits =
#  int_of_big_int (ceiling_ratio (mult_int_ratio (digits, #[0.1038103])))
#  ;;
Value num_of_digits : int -> int

#let approx_pi10 digits = approx_pi (num_of_digits digits)
#  ;;
Value approx_pi10 : int -> ratio

##use (lib_directory ^ "format_numbers");;
Directive : unit

```

```

#let show_pi digits =
# let s = approx_ratio_fix (digits, approx_pi10 digits) in
# let len = sub_int (length_string s, #3) in
# let v = vector #1 of (sub_string s #3 len) in
# let res = beautiful_string_vect_of_string_vect
#   (v, #0, #0, big_int_of_int len) in
# print_string "+3.";print_newline();
# for i = #0 to pred_int (vect_length res) do
#   print_string res.(i)
# done
#;;
Value show_pi : int -> unit

```

A.6.4 Running the program

```

#timers true;;
: unit

```

```

#echo_values false;;
: unit

```

```

#approx_pi10 10000;;
: ratio
Evaluation has needed: Runtime: 36.68s

```

```

#approx_pi10 20000;;
: ratio
Evaluation has needed: Runtime: 155.06s GC: 0.95s

```

And now a beautiful printing of 2000 digits (contained quite exactly in one page):

```

#show_pi 2000;;
+3.
14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679
82148 08651 32823 06647 09384 46095 50582 23172 53594 08128
48111 74502 84102 70193 85211 05559 64462 29489 54930 38196
44288 10975 66593 34461 28475 64823 37867 83165 27120 19091

45648 56692 34603 48610 45432 66482 13393 60726 02491 41273
72458 70066 06315 58817 48815 20920 96282 92540 91715 36436
78925 90360 01133 05305 48820 46652 13841 46951 94151 16094
33057 27036 57595 91953 09218 61173 81932 61179 31051 18548
07446 23799 62749 56735 18857 52724 89122 79381 83011 94912

98336 73362 44065 66430 86021 39494 63952 24737 19070 21798
60943 70277 05392 17176 29317 67523 84674 81846 76694 05132
00056 81271 45263 56082 77857 71342 75778 96091 73637 17872
14684 40901 22495 34301 46549 58537 10507 92279 68925 89235
42019 95611 21290 21960 86403 44181 59813 62977 47713 09960

51870 72113 49999 99837 29780 49951 05973 17328 16096 31859
50244 59455 34690 83026 42522 30825 33446 85035 26193 11881
71010 00313 78387 52886 58753 32083 81420 61717 76691 47303
59825 34904 28755 46873 11595 62863 88235 37875 93751 95778
18577 80532 17122 68066 13001 92787 66111 95909 21642 01989

38095 25720 10654 85863 27886 59361 53381 82796 82303 01952
03530 18529 68995 77362 25994 13891 24972 17752 83479 13151
55748 57242 45415 06959 50829 53311 68617 27855 88907 50983
81754 63746 49393 19255 06040 09277 01671 13900 98488 24012
85836 16035 63707 66010 47101 81942 95559 61989 46767 83744

94482 55379 77472 68471 04047 53464 62080 46684 25906 94912
93313 67702 89891 52104 75216 20569 66024 05803 81501 93511
25338 24300 35587 64024 74964 73263 91419 92726 04269 92279
67823 54781 63600 93417 21641 21992 45863 15030 28618 29745
55706 74983 85054 94588 58692 69956 90927 21079 75093 02955

32116 53449 87202 75596 02364 80665 49911 98818 34797 75356
63698 07426 54252 78625 51818 41757 46728 90977 77279 38000
81647 06001 61452 49192 17321 72147 72350 14144 19735 68548
16136 11573 52552 13347 57418 49468 43852 33239 07394 14333
45477 62416 86251 89835 69485 56209 92192 22184 27255 02542

56887 67179 04946 01653 46680 49886 27232 79178 60857 84383
82796 79766 81454 10095 38837 86360 95068 00642 25125 20511
73929 84896 08412 84886 26945 60424 19652 85022 21066 11863
06744 27862 20391 94945 04712 37137 86960 95636 43719 17287
46776 46575 73962 41389 08658 32645 99581 33904 78027 59010
: unit
Evaluation has needed: Runtime: 1.96s

```

Annexe B

To normalize or not to normalize rational numbers?

In this appendix, we try to answer the two following questions:

- when is it necessary to normalize,
- is it on average a good strategy always to normalize or not to normalize.

B.1 On average not to normalize is the better strategy

Let p (resp. p_n) be the probability for two integers a and b , to be mutually prime (resp. to have a gcd equal to n). Plainly, $\gcd(a, b) = n$ if and only if $a \bmod n = 0$ and $b \bmod n = 0$ and

$$\gcd\left(\frac{a}{n}, \frac{b}{n}\right) = 1.$$

Thus

$$p_n = \frac{1}{n} \times \frac{1}{n} \times p.$$

Hence we have :

$$1 = \sum_{n=1}^{n=\infty} \frac{p}{n^2} = p \times \sum_{n=1}^{n=\infty} \frac{1}{n^2} = p \frac{\pi^2}{6}$$

thus

$$p = \frac{6}{\pi^2}$$

and

$$p_n = \frac{6}{\pi^2 n^2}.$$

Numerically speaking p is greater than 60% and the probability that two integers have gcd less than or equal to 10 is greater than 94%.

So 60% of normalization of rational numbers are absolutely without effect and 94% will need a long time and simplify the rational number by a very small factor compared to the size of the rational number.

This viewpoint guides our choice of the default position for the "normalize ratio when computing" flag (see section 8.6 of this manual).

But now we will see an example where it is much faster to normalize after each step of the computation than not to normalize.

B.2 Exception

B.2.1 Exposition

Here is the origin of this sequence. Jean-Michel Muller in [4] illustrates the problem of round off errors with it. This sequence converges slowly to 6 but with non exact rational arithmetic it seems to converges fastly to 100. In fact, for computing one term, it is much faster to normalize the term after each step, since successive gcd of no normalized numerator and denominator are much bigger than the normalized numerator and denominator.

And now here is the definition of this sequence:

$$\begin{aligned} a_0 &= \frac{11}{2} \\ a_1 &= \frac{61}{11} \\ a_{n+1} &= 111 - \frac{1130 - \frac{3000}{a_{n-1}}}{a_n} \end{aligned}$$

In a closed form,

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}.$$

B.2.2 Floating point version

To convince the reader of the different behaviour when using floating point arithmetic, here is

```
##open arith;;
Pragma : unit

##arith float;;
Symbol prefix + overloaded with add_float : float * float -> float
Symbol prefix * overloaded with mult_float : float * float -> float
Symbol prefix ** overloaded with power_float : float * float -> float
Symbol prefix - overloaded with sub_float : float * float -> float
Symbol prefix / overloaded with div_float : float * float -> float
Symbol prefix <= overloaded with le_float : float * float -> bool
Symbol prefix < overloaded with lt_float : float * float -> bool
Symbol prefix >= overloaded with ge_float : float * float -> bool
Symbol prefix > overloaded with gt_float : float * float -> bool
Symbol - overloaded with minus_float : float -> float
Symbol pred overloaded with pred_float : float -> float
Symbol succ overloaded with succ_float : float -> float
Symbol incr overloaded with incr_float : float ref -> float
Symbol decr overloaded with decr_float : float ref -> float
```

```

Symbol max overloaded with max_float : float -> float -> float
Symbol min overloaded with min_float : float -> float -> float
Symbol abs overloaded with abs_float : float -> float
Symbol prefix power overloaded with power_float : float * float -> float

```

```

#let a n =
# let rec arec = function
#   0 -> (11.0/2.0,61.0/11.0)
# | n -> let (x, y) = arec (pred_int n) in
#       (y, (111.0 - (1130.0 - 3000.0/x)/y))
#in snd (arec (pred_int n))
#;;
Value a : int -> float

#a 20;;
: float

```

B.2.3 The program

This sequence can be implemented as follows:

```

##open arith;;
Pragma : unit

##arith ratio;;
Symbol prefix + overloaded with add_ratio : ratio * ratio -> ratio
Symbol prefix * overloaded with mult_ratio : ratio * ratio -> ratio
Symbol prefix - overloaded with sub_ratio : ratio * ratio -> ratio
Symbol prefix / overloaded with div_ratio : ratio * ratio -> ratio
Symbol prefix <= overloaded with le_ratio : ratio * ratio -> bool
Symbol prefix < overloaded with lt_ratio : ratio * ratio -> bool
Symbol prefix >= overloaded with ge_ratio : ratio * ratio -> bool
Symbol prefix > overloaded with gt_ratio : ratio * ratio -> bool
Symbol - overloaded with minus_ratio : ratio -> ratio
Symbol float overloaded with float_of_ratio : ratio -> float
Symbol max overloaded with max_ratio : ratio * ratio -> ratio
Symbol min overloaded with min_ratio : ratio * ratio -> ratio
Symbol abs overloaded with abs_ratio : ratio -> ratio

#let a n =
# let rec arec = function
#   #0 -> ([11/2],[61/11])
# | n -> let (x, y) = arec (pred_int n) in
#       (y, ([111] - ([1130] - #[3000]/x)/y))
#in snd (arec (pred_int n))
#;;
Value a : int -> ratio

```

and we compare the runtime for a_{20} with and without normalizing during computation:

```

#timers true;;
: unit

```

```
#set_normalize_ratio true;;
: bool

#a 20;;
: ratio
Evaluation has needed: Runtime: 0.10s
```

```
#approx_ratio_fix (#3, it);;
: string
```

```
#set_normalize_ratio false;;
: bool
```

```
#a 20;;
: ratio
Evaluation has needed: Runtime: 4.12s
```

and more outstandingly for a_{30} we obtain:

```
#set_normalize_ratio true;;
: bool
```

```
#a 30;;
: ratio
Evaluation has needed: Runtime: 0.17s GC: 1.00s
```

```
#approx_ratio_fix (#3, it);;
: string
```

```
#set_normalize_ratio false;;
: bool
```

```
#a 30;;
```

```
Evaluation Failed: failure "create_nat: number too long"
Evaluation has needed: Runtime: 29.01s GC: 1.89s
```

We see on this example that this is a very small rational number (its numerator and denominator have only three 2^{32} -digits). If we do not perform normalization during computation, it is represented by two integers with more than 8191 2^{32} -digits. This proves well how normalizing during computation may be important in exceptional cases.

So we recommend in practice to try both versions. If a rational computation seems particularly slow, maybe you should turn this flag on and try again.

Bibliographie

- [1] D. V. CHUDNOVSKY AND G. V. CHUDNOVSKY. The computation of classical constants. *Proc. Natl. Acad. Sci. USA* 86 (November 1989), 8178–8182.
- [2] J.-C. HERVÉ, F. MORAIN, D. SALESIN, B. SERPETTE, J. VUILLEMIN, AND P. ZIMMERMANN. Bignum: A portable and efficient package for arbitrary-precision arithmetic. Tech. Rep. 1016, INRIA, Domaine de Voluceau, 78153 Rocquencourt, FRANCE, April 1989.
- [3] D. E. KNUTH. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1981.
- [4] J. M. MULLER. *Arithmétique des ordinateurs*. Masson, 1989.

Concept index

†	19, 40
*	19
#	27, 36
*	23, 34, 71, 83, 96
**	31
+	23, 34, 71, 83, 96
-	23, 34, 71, 83, 96, 110
/	23, 34, 71, 83, 96
<	25, 34, 72, 86, 99
<=	25, 34, 72, 86, 99
=	25, 34, 72, 86, 99
>	25, 34, 72, 86, 99
>=	25, 34, 72, 86, 99

A

abs	25, 34, 72, 86, 99
#arith	110
#arith cautious	110

B

big_int	69
---------	----

C

ceiling	97
---------	----

coercion between <code>ints</code> and <code>nums</code>	99
coercion between types <code>big_int</code> and <code>num</code>	101
coercion between types <code>big_int</code> and <code>ratio</code>	88
coercion between types <code>big_int</code> and <code>string</code>	74, 76
coercion between types <code>float</code> and <code>big_int</code>	73
coercion between types <code>float</code> and <code>nat</code>	60
coercion between types <code>float</code> and <code>num</code>	100
coercion between types <code>float</code> and <code>ratio</code>	88
coercion between types <code>float</code> and <code>string</code>	36, 37
coercion between types <code>int</code> and <code>big_int</code>	73
coercion between types <code>int</code> and <code>char</code>	27
coercion between types <code>int</code> and <code>float</code>	26
coercion between types <code>int</code> and <code>nat</code>	59
coercion between types <code>int</code> and <code>ratio</code>	87
coercion between types <code>int</code> and <code>string</code>	27, 29
coercion between types <code>nat</code> and <code>num</code>	100
coercion between types <code>nat</code> and <code>ratio</code>	87
coercion between types <code>nat</code> and <code>string</code>	61, 61
coercion between types <code>ratio</code> and <code>num</code>	101
coercion between types <code>string</code> and <code>num</code>	102, 103
coercion between types <code>string</code> and <code>ratio</code>	89, 90

D

<code>decr</code>	22, 34, 96
<code>denominator</code>	95

F

<code>#fast arith</code>	109
<code>float</code>	26, 100
<code>float</code>	33
floating point numbers	33
<code>floor</code>	97

G

gcd..... 23, 71

I

incr..... 22, 34, 96

int..... 21, 27

integer..... 21, 27, 69

integer..... 97

is_integer..... 97

is_integer_num..... 99

is_integer_ratio..... 87

M

max..... 25, 34, 72, 86, 99

min..... 25, 34, 72, 86, 99

monster_int..... 23, 25, 72

N

nat..... 39

num..... 95

numerator..... 95, 97

O

#open arith..... 110

overload ... with 110

P

pred..... 22, 33, 70, 96

print..... 29, 37, 61

printing..... 29, 37, 61

R

ratio..... 81

rational numbers..... 81

round..... 97

S

#standard arith	109
succ	22, 33, 70, 96

Function index

Note the meaning of the two following symbols placed beyond the name of an arithmetic function :

† this function is only available in open arithmetic mode,

* this function is one of the kernel part of the BigNum

A

abs : int → int	110
abs_big_int : big_int → big_int	72
abs_float : float → float	34
abs_int : int → int	25
abs_num : num → num	98
abs_ratio : ratio → ratio	86
acos : float → float	35
add_big_int : big_int × big_int → big_int	70
add_big_int_ratio [†] : big_int × ratio → ratio	83
add_float : float × float → float	34
add_int : int × int → int	23
add_int_big_int [†] : int × big_int → big_int	70
add_int_ratio [†] : int × ratio → ratio	83
add_nat ^{†*} : nat × int × int × nat × int × int × int → int	44
add_num : num × num → num	96
add_ratio : ratio × ratio → ratio	83
approx_num_exp : int × num → string	106
approx_num_fix : int × num → string	106
approx_ratio_exp [†] : int × ratio → string	93
approx_ratio_fix [†] : int × ratio → string	93

arith_status : unit → unit	112
asin : float → float	35
atan : float → float	35

B

base_digit_of_char [†] : char × int → int	61
base_power_big_int [†] : int × int × big_int → big_int	78
big_int_of_float [†] : float → big_int	73
big_int_of_int [†] : int → big_int	73
big_int_of_nat [†] : nat → big_int	74
big_int_of_num [†] : num → big_int	101
big_int_of_ratio [†] : ratio → big_int	88
big_int_of_string [†] : string → big_int	74
biggest_int : int	21
blit_nat ^{†*} : nat × int × nat × int × int → unit	41

C

cautious_normalize_num_when_printing [†] : num → num	95
cautious_normalize_ratio [†] : ratio → ratio	81
cautious_normalize_ratio_when_printing [†] : ratio → ratio	81
ceiling_num [†] : num → num	97
ceiling_ratio [†] : ratio → big_int	84
char_of_int : int → char	27
compare_big_int [†] : big_int × big_int → int	71
compare_big_int_ratio [†] : big_int × ratio → int	86
compare_digits_nat : nat × int × nat × int → int	57
compare_int [†] : int × int → int	25
compare_nat [†] : nat × int × int × nat × int × int → int	57
compare_num [†] : num × num → int	98
compare_ratio [†] : ratio × ratio → int	86
complement_nat ^{†*} : nat × int × int → unit	47
copy_nat [†] : nat × int × int → nat	40
cos : float → float	35

create_big_int [†] : $\text{int} \times \text{nat} \rightarrow \text{big_int}$	69
create_nat [†] : $\text{int} \rightarrow \text{nat}$	40
create_normalized_ratio [†] : $\text{big_int} \times \text{big_int} \rightarrow \text{ratio}$	82
create_ratio [†] : $\text{big_int} \times \text{big_int} \rightarrow \text{ratio}$	82

D

debug_print_nat [†] : $\text{nat} \rightarrow \text{unit}$	61
debug_string_nat [†] : $\text{nat} \rightarrow \text{string}$	61
debug_string_vect_nat [†] : $\text{nat} \rightarrow \text{string vect}$	61
decimal_of_string [†] : $\text{int} \times \text{string} \times \text{int} \times \text{int} \rightarrow \text{string} \times \text{int}$	74
decr : $\text{int ref} \rightarrow \text{int}$	110
decr_float : $\text{float ref} \rightarrow \text{float}$	34
decr_int : $\text{int ref} \rightarrow \text{int}$	22
decr_nat ^{†*} : $\text{nat} \times \text{int} \times \text{int} \times \text{int} \rightarrow \text{int}$	47
decr_num : $\text{num ref} \rightarrow \text{num}$	96
denominator_num [†] : $\text{num} \rightarrow \text{num}$	95
denominator_ratio [†] : $\text{ratio} \rightarrow \text{big_int}$	82
display_float : $\text{float} \rightarrow \text{unit}$	37
display_int : $\text{int} \rightarrow \text{unit}$	29
display_num : $\text{num} \rightarrow \text{unit}$	103
div_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$	71
div_big_int_ratio [†] : $\text{big_int} \times \text{ratio} \rightarrow \text{ratio}$	83
div_digit_nat ^{†*} : $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{int}$	54
div_float : $\text{float} \times \text{float} \rightarrow \text{float}$	34
div_int_ratio [†] : $\text{int} \times \text{ratio} \rightarrow \text{ratio}$	83
div_nat [†] : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{unit}$	54
div_num : $\text{num} \times \text{num} \rightarrow \text{num}$	96
div_ratio : $\text{ratio} \times \text{ratio} \rightarrow \text{ratio}$	83
div_ratio_big_int [†] : $\text{ratio} \times \text{big_int} \rightarrow \text{ratio}$	83
div_ratio_int [†] : $\text{ratio} \times \text{int} \rightarrow \text{ratio}$	83

E

echo_num : $\text{num} \rightarrow \text{unit}$	103
eq_big_int [†] : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$	71

eq_big_int_ratio [†] : $\text{big_int} \times \text{ratio} \rightarrow \text{bool}$	86
eq_float : $\text{float} \times \text{float} \rightarrow \text{bool}$	34
eq_int : $\text{int} \times \text{int} \rightarrow \text{bool}$	25
eq_nat [†] : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{bool}$	57
eq_num : $\text{num} \times \text{num} \rightarrow \text{bool}$	98
eq_ratio [†] : $\text{ratio} \times \text{ratio} \rightarrow \text{bool}$	86
exp : $\text{float} \rightarrow \text{float}$	35

F

float : $\text{int} \rightarrow \text{float}$	110
float_num [†] : $\text{num} \rightarrow \text{num}$	106
float_of_big_int [†] : $\text{big_int} \rightarrow \text{float}$	73
float_of_int : $\text{int} \rightarrow \text{float}$	26
float_of_nat : $\text{nat} \rightarrow \text{float}$	60
float_of_num : $\text{num} \rightarrow \text{float}$	100
float_of_ratio [†] : $\text{ratio} \rightarrow \text{float}$	88
float_of_rational_string [†] : $\text{ratio} \rightarrow \text{string}$	93
float_of_string : $\text{string} \rightarrow \text{float}$	36
floor_num [†] : $\text{num} \rightarrow \text{num}$	97
floor_ratio [†] : $\text{ratio} \rightarrow \text{big_int}$	84

G

gcd_big_int [†] : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$	71
gcd_int [†] : $\text{int} \times \text{int} \rightarrow \text{int}$	23
gcd_int_nat [†] : $\text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$	54
gcd_nat [†] : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{int}$	54
ge_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$	71
ge_big_int_ratio [†] : $\text{big_int} \times \text{ratio} \rightarrow \text{bool}$	86
ge_float : $\text{float} \times \text{float} \rightarrow \text{bool}$	34
ge_int : $\text{int} \times \text{int} \rightarrow \text{bool}$	25
ge_nat [†] : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{bool}$	57
ge_num : $\text{num} \times \text{num} \rightarrow \text{bool}$	98
ge_ratio : $\text{ratio} \times \text{ratio} \rightarrow \text{bool}$	86
get_approx_printing : $\text{unit} \rightarrow \text{bool}$	112

get_arith_overloading : unit → arith list	110
get_error_when_null_denominator : unit → bool	112
get_floating_precision : unit → int	112
get_normalize_ratio [†] : unit → bool	81, 111
get_normalize_ratio_when_printing [†] : unit → bool	81, 111
gt_big_int : big_int × big_int → bool	71
gt_big_int_ratio [†] : big_int × ratio → bool	86
gt_float : float × float → bool	34
gt_int : int × int → bool	25
gt_nat [†] : nat × int × int × nat × int × int → bool	57
gt_num : num × num → bool	98
gt_ratio : ratio × ratio → bool	86

I

incr : int ref → int	110
incr_float : float ref → float	34
incr_int : int ref → int	22
incr_nat ^{†*} : nat × int × int × int → int	44
incr_num : num ref → num	96
int_of_big_int [†] : big_int → int	73
int_of_char : char → int	27
int_of_float : float → int	26
int_of_nat [†] : nat → int	59
int_of_num : num → int	99
int_of_ratio [†] : ratio → int	87
int_of_string : string → int	28
integer_num [†] : num → num	97
integer_ratio [†] : ratio → big_int	84
inverse_ratio [†] : ratio → ratio	83
is_digit_int [†] : nat × int → bool	42
is_digit_normalized ^{†*} : nat × int → bool	42
is_digit_odd [†] : nat × int → bool	42
is_digit_zero ^{†*} : nat × int → bool	42
is_int : num → bool	99

is_int_big_int [†] : $\text{big_int} \rightarrow \text{bool}$	73
is_integer_num [†] : $\text{num} \rightarrow \text{bool}$	97
is_integer_ratio [†] : $\text{ratio} \rightarrow \text{bool}$	84
is_nat_int [†] : $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{bool}$	42
is_normalized_ratio [†] : $\text{ratio} \rightarrow \text{bool}$	81
is_zero_nat [†] : $\text{nat} \times \text{int} \times \text{int} \rightarrow \text{bool}$	42

L

land_digit_nat : $\text{nat} \times \text{int} \times \text{nat} \times \text{int} \rightarrow \text{unit}$	65
land_int : $\text{int} \times \text{int} \rightarrow \text{int}$	30
latex_print_big_int [†] : $\text{big_int} \rightarrow \text{unit}$	77
latex_print_for_read_big_int [†] : $\text{big_int} \rightarrow \text{unit}$	77
latex_print_for_read_nat [†] : $\text{nat} \rightarrow \text{unit}$	64
latex_print_for_read_num [†] : $\text{num} \rightarrow \text{unit}$	104
latex_print_for_read_ratio [†] : $\text{ratio} \rightarrow \text{unit}$	91
latex_print_nat [†] : $\text{nat} \rightarrow \text{unit}$	64
latex_print_num [†] : $\text{num} \rightarrow \text{unit}$	104
latex_print_ratio [†] : $\text{ratio} \rightarrow \text{unit}$	91
le_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{bool}$	71
le_big_int_ratio [†] : $\text{big_int} \times \text{ratio} \rightarrow \text{bool}$	86
le_float : $\text{float} \times \text{float} \rightarrow \text{bool}$	34
le_int : $\text{int} \times \text{int} \rightarrow \text{bool}$	25
le_nat [†] : $\text{nat} \times \text{int} \times \text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{bool}$	57
le_num : $\text{num} \times \text{num} \rightarrow \text{bool}$	98
le_ratio : $\text{ratio} \times \text{ratio} \rightarrow \text{bool}$	86
leading_digit_big_int [†] : $\text{big_int} \rightarrow \text{int}$	78
least_int : int	21
length_nat [†] : $\text{nat} \rightarrow \text{int}$	42
length_of_digit [†] : int	39
length_of_int [†] : int	21
lnot_int : $\text{int} \rightarrow \text{int}$	30
log : $\text{float} \rightarrow \text{float}$	35
log10 : $\text{float} \rightarrow \text{float}$	35

lor_digit_nat : nat × int × nat × int → unit	65
lor_int : int × int → int	30
lshift_int : int × int → int	30
lt_big_int : big_int × big_int → bool	71
lt_big_int_ratio [†] : big_int × ratio → bool	86
lt_float : float × float → bool	34
lt_int : int × int → bool	25
lt_nat [†] : nat × int × int × nat × int × int → bool	57
lt_num : num × num → bool	98
lt_ratio : ratio × ratio → bool	86
lxor_digit_nat : nat × int × nat × int → unit	65
lxor_int : int × int → int	30

M

make_nat [†] : int → nat	40
max : int → int → int	110
max_big_int : big_int × big_int → big_int	72
max_float : float → float → float	34
max_int : int → int → int	25
max_num : num → num → num	99
max_ratio : ratio × ratio → ratio	86
min : int → int → int	110
min_big_int : big_int × big_int → big_int	72
min_float : float → float → float	34
min_int : int → int → int	25
min_num : num → num → num	99
min_ratio : ratio × ratio → ratio	86
minus_big_int : big_int → big_int	70
minus_float : float → float	34
minus_int : int → int	23, 25
minus_num : num → num	96
minus_ratio : ratio → ratio	83
mod_big_int : big_int × big_int → big_int	71
mod_int : int × int → int	23

mod_num : num × num → num	96
monster_int : int	21, 22
msd_ratio [†] : ratio → int	93
mult_big_int : big_int × big_int → big_int	70
mult_big_int_ratio [†] : big_int × ratio → ratio	83
mult_digit_nat : nat × int × int × nat × int × int × nat × int → int	50
mult_float : float × float → float	34
mult_int : int × int → int	23
mult_int_big_int [†] : int × big_int → big_int	70
mult_int_ratio [†] : int × ratio → ratio	83
mult_nat : nat × int × int × nat × int × int × nat × int × int → int	50
mult_num : num × num → num	96
mult_ratio : ratio × ratio → ratio	83

N

nat_of_big_int [†] : big_int → nat	74
nat_of_float : float → nat	60
nat_of_int : int → nat	59
nat_of_num [†] : num → nat	100
nat_of_ratio [†] : ratio → nat	87
nat_of_string : string → nat	61
normalize_num [†] : num → num	95
normalize_ratio [†] : ratio → ratio	81
nth_digit_big_int [†] : big_int × big_int → int	78
nth_digit_nat ^{†*} : nat × int → int	42
null_denominator [†] : ratio → bool	82
num_bits_int [†] : int → int	31
num_decimal_digits_int [†] : int → int	31
num_digits_big_int [†] : big_int → int	78
num_digits_nat ^{†*} : nat × int × int → int	42
num_leading_zero_bits_in_digit ^{†*} : nat × int → int	42
num_of_big_int [†] : big_int → num	101
num_of_float : float → num	100
num_of_int : int → num	99

num_of_nat [†] : nat → num.....	100
num_of_ratio [†] : ratio → num.....	101
num_of_string : string → num.....	102
numerator_num [†] : num → num.....	95
numerator_ratio [†] : ratio → big_int.....	82

P

power_base_int [†] : int × int → nat.....	66
power_base_nat [†] : int × nat × int × int → nat.....	66
power_big_int_positive_big_int [†] : big_int × big_int → big_int.....	78
power_big_int_positive_int [†] : big_int × int → big_int.....	78
power_float : float × float → float.....	35
power_int : int × int → int.....	31
power_int_positive_big_int [†] : int × big_int → big_int.....	78
power_int_positive_int [†] : int × int → big_int.....	31, 78
power_num : num × num → num.....	106
power_num_big_int [†] : num × big_int → num.....	106
power_num_int [†] : num × int → num.....	106
power_ratio_positive_big_int [†] : ratio × big_int → ratio.....	92
power_ratio_positive_int [†] : ratio × int → ratio.....	92
pred : int → int.....	110
pred_big_int : big_int → big_int.....	70
pred_float : float → float.....	33
pred_int : int → int.....	22
pred_num : num → num.....	96
prefix < : int × int → bool.....	110
prefix <= : int × int → bool.....	110
prefix > : int × int → bool.....	110
prefix >= : int × int → bool.....	110
prefix * : int × int → int.....	110
prefix ** : int × int → int.....	110
prefix + : int × int → int.....	110
prefix - : int × int → int.....	110
prefix / : int × int → int.....	110

prefix mod : $\text{int} \times \text{int} \rightarrow \text{int}$	110
prefix power : $\text{int} \times \text{int} \rightarrow \text{int}$	111
prefix quo : $\text{int} \times \text{int} \rightarrow \text{int}$	110
print_beautiful_big_int [†] : $\text{big_int} \rightarrow \text{unit}$	77
print_beautiful_nat [†] : $\text{nat} \rightarrow \text{unit}$	64
print_beautiful_num [†] : $\text{num} \rightarrow \text{unit}$	104
print_beautiful_ratio [†] : $\text{ratio} \rightarrow \text{unit}$	91
print_big_int [†] : $\text{big_int} \rightarrow \text{unit}$	76
print_big_int_for_read : $\text{big_int} \rightarrow \text{unit}$	76
print_float : $\text{float} \rightarrow \text{unit}$	37
print_float_for_read : $\text{float} \rightarrow \text{unit}$	37
print_int : $\text{int} \rightarrow \text{unit}$	29
print_int_for_read : $\text{int} \rightarrow \text{unit}$	29
print_nat [†] : $\text{nat} \rightarrow \text{unit}$	61
print_nat_for_read [†] : $\text{nat} \rightarrow \text{unit}$	61
print_num : $\text{num} \rightarrow \text{unit}$	103
print_num_for_read : $\text{num} \rightarrow \text{unit}$	103
print_ratio [†] : $\text{ratio} \rightarrow \text{unit}$	90
print_ratio_for_read [†] : $\text{ratio} \rightarrow \text{unit}$	90

Q

quo_big_int : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int}$	71
quo_int : $\text{int} \times \text{int} \rightarrow \text{int}$	23
quo_num : $\text{num} \times \text{num} \rightarrow \text{num}$	96
quomod_big_int [†] : $\text{big_int} \times \text{big_int} \rightarrow \text{big_int} \times \text{big_int}$	71

R

ratio_of_big_int [†] : $\text{big_int} \rightarrow \text{ratio}$	88
ratio_of_float [†] : $\text{float} \rightarrow \text{ratio}$	88
ratio_of_int [†] : $\text{int} \rightarrow \text{ratio}$	87
ratio_of_nat [†] : $\text{nat} \rightarrow \text{ratio}$	87
ratio_of_num [†] : $\text{num} \rightarrow \text{ratio}$	101
ratio_of_string [†] : $\text{string} \rightarrow \text{ratio}$	89

report_sign_ratio [†] : ratio × big_int → big_int	86
round_futur_last_digit [†] : string × int × int → bool	93
round_num [†] : num → num	97
round_ratio [†] : ratio → big_int	84

S

set_approx_printing : bool → bool	112
set_digit_nat ^{†*} : nat × int × int → unit	41
set_error_when_null_denominator : bool → bool	112
set_floating_precision : int → int	112
set_length_of_digit [†] : int → unit	39
set_normalize_ratio [†] : bool → bool	81, 111
set_normalize_ratio_when_printing [†] : bool → bool	81, 111
set_to_zero_nat ^{†*} : nat × int × int → unit	41
shift_left_nat : nat × int × int × nat × int × int → unit	50
shift_right_nat ^{†*} : nat × int × int × nat × int × int → unit	54
sign_big_int [†] : big_int → int	72
sign_int [†] : int → int	25
sign_num [†] : num → int	98
sign_ratio [†] : ratio → int	86
simple_big_int_of_string [†] : int × string × int × int → big_int	74
sin : float → float	35
sqrt : float → float	35
sqrt_big_int [†] : big_int → big_int	78
sqrt_nat [†] : nat × int × int → nat	66
square_big_int [†] : big_int → big_int	70
square_nat : nat × int × int × nat × int × int → int	50
square_num [†] : num → num	96
square_ratio [†] : ratio → ratio	83
string_for_read_of_big_int [†] : big_int → string	76
string_for_read_of_float : float → string	37
string_for_read_of_int : int → string	29
string_for_read_of_nat [†] : nat → string	61

string_for_read_of_num : num \rightarrow string	103
string_for_read_of_ratio [†] : ratio \rightarrow string	90
string_of_big_int [†] : big_int \rightarrow string	76
string_of_digit [†] : nat \rightarrow string	61
string_of_float : float \rightarrow string	37
string_of_int : int \rightarrow string	29
string_of_nat [†] : nat \rightarrow string	61
string_of_num : num \rightarrow string	103
string_of_ratio [†] : ratio \rightarrow string	90
sub_big_int : big_int \times big_int \rightarrow big_int	70
sub_float : float \times float \rightarrow float	34
sub_int : int \times int \rightarrow int	23
sub_nat ^{†*} : nat \times int \times int \times nat \times int \times int \times int \rightarrow int	47
sub_num : num \times num \rightarrow num	96
sub_ratio : ratio \times ratio \rightarrow ratio	83
succ : int \rightarrow int	110
succ_big_int : big_int \rightarrow big_int	70
succ_float : float \rightarrow float	33
succ_int : int \rightarrow int	22
succ_num : num \rightarrow num	96
sys_big_int_of_string [†] : int \times string \times int \times int \rightarrow big_int	74
sys_float_of_nat [†] : nat \times int \times int \rightarrow float	60
sys_float_of_string : int \times string \times int \times int \rightarrow float	36
sys_int_of_nat : nat \times int \times int \rightarrow int	59
sys_int_of_string [†] : int \times string \times int \times int \rightarrow int	28
sys_latex_print_big_int [†] : int \times string \times big_int \times string \rightarrow unit	77
sys_latex_print_nat [†] : int \times string \times nat \times int \times int \times string \rightarrow unit	64
sys_latex_print_num [†] : int \times string \times num \times string \rightarrow unit	104
sys_latex_print_ratio [†] : int \times string \times ratio \times string \rightarrow unit	91
sys_nat_of_string : int \times string \times int \times int \rightarrow nat	61
sys_num_of_string [†] : int \times string \times int \times int \rightarrow num	102
sys_print_beautiful_big_int [†] : α \times string \times big_int \times string \rightarrow unit	77
sys_print_beautiful_nat [†] : int \times string \times nat \times int \times int \times string \rightarrow unit	64

<code>sys_print_beautiful_num</code> [†] : $\text{int} \times \text{string} \times \text{num} \times \text{string} \rightarrow \text{unit}$	104
<code>sys_print_beautiful_ratio</code> [†] : $\text{int} \times \text{string} \times \text{ratio} \times \text{string} \rightarrow \text{unit}$	91
<code>sys_print_big_int</code> [†] : $\text{int} \times \text{string} \times \text{big_int} \times \text{string} \rightarrow \text{unit}$	76
<code>sys_print_nat</code> [†] : $\text{int} \times \text{string} \times \text{nat} \times \text{int} \times \text{int} \times \text{string} \rightarrow \text{unit}$	61
<code>sys_print_num</code> [†] : $\text{int} \times \text{string} \times \text{num} \times \text{string} \rightarrow \text{unit}$	103
<code>sys_print_ratio</code> [†] : $\text{int} \times \text{string} \times \text{ratio} \times \text{string} \rightarrow \text{unit}$	90
<code>sys_ratio_of_string</code> [†] : $\text{int} \times \text{string} \times \text{int} \times \text{int} \rightarrow \text{ratio}$	89
<code>sys_string_list_of_nat</code> [†] : $\text{int} \times \text{nat} \times \text{int} \times \text{int} \rightarrow \text{string list}$	61
<code>sys_string_of_big_int</code> : $\text{int} \times \text{string} \times \text{big_int} \times \text{string} \rightarrow \text{string}$	76
<code>sys_string_of_digit</code> [†] : $\text{nat} \times \text{int} \rightarrow \text{string}$	61
<code>sys_string_of_int</code> [†] : $\text{int} \times \text{string} \times \text{int} \times \text{string} \rightarrow \text{string}$	29
<code>sys_string_of_nat</code> : $\text{int} \times \text{string} \times \text{nat} \times \text{int} \times \text{int} \times \text{string} \rightarrow \text{string}$	61
<code>sys_string_of_num</code> [†] : $\text{int} \times \text{string} \times \text{num} \times \text{string} \rightarrow \text{string}$	103
<code>sys_string_of_ratio</code> [†] : $\text{int} \times \text{string} \times \text{ratio} \times \text{string} \rightarrow \text{string}$	90

V

<code>verify_null_denominator</code> [†] : $\text{ratio} \rightarrow \text{bool}$	82
--	-----------

Z

<code>zero_big_int</code> [†] : big_int	72
--	-----------