Partial Documentation

genarith

2, avenue Galliéni
B.P. 85
94250 Gentilly Cedex
FRANCE

Version of December 2, 1993

0.1 Generic Number Arithmetic

This section describes the use of generic arithmetic operations for numbers. Generic arithmetic operations for non-numeric datatypes, such as `<time>`, are described in the sections devoted to those datatypes.

0.1.1 Type predicates and conversions

`(numberp` *n*`)` [*function*]

Summary Determines if an object is a number.

Arguments *n* `<object>`

Result `<number> or <null>`

This function determines whether *n* is a number. If it is, `numberp` returns *n*. If not, it returns `()`.

```
? (numberp 12)
= 12
? (numberp 3.1)
= 3.1
? (numberp 2/3)
= 2/3
? (numberp 'three-point-one)
= ()
```

See also `integerp` (??) `floatp` (??)

`(integer` *x*`)` [*generic function*]

Summary Conversion to integer.

Arguments *x* `<object>`

Result `<integer>`

Allocation `<big-int>` if needed

If the argument *x* is an integer, the generic function `integer` returns it as its value. If *x* is a floating-point number, its integer part is returned — in other words, it is truncated. If *x* is a ratio, it is converted to a floating-point number and truncated. If the argument is not a number, a type-specific operation is applied to retrieve an integer. New methods may be defined for your own datatypes if instances of them can be converted naturally into an integer.

```
? (integer 123)
= 123
? (integer 10.4)
= 10
? (integer -10.4)
= -10
? (integer 1.234e+20)
= 123400000000000000000
? (integer 10/3)
= 3
```

See also `int` (3) `floor` (3) `ceiling` (3) `integerp` (??)

(int *x*) [*function*]

Summary Conversion to an int.

Arguments *x* `<number>`

Result `<int>`

This function is identical to `integer`, but always return an int; it signals an error if *x* cannot be converted to an int.

```
? (int 123)
= 123
? (int 10.4)
= 10
? (int -10.4)
= -10
? (int 1.234e+20)
 ** int: '1.234e+20' cannot be converted to an int
```

See also `int` (3) `floor` (3) `ceiling` (3) `integerp` (??)

(floor *x*) [*function*]

(ceiling *x*) [*function*]

(round *x*) [*function*]

Summary Rounding.

Arguments *x* `<number>`

Result `<integer>`

Allocation `<float>` and `<big-int>` if needed

If the argument *x* is an integer, these functions return it. Otherwise, *x* is converted into an integer using one of the following rounding methods:

- `floor`: This conversion is a "downwards" truncation. Its value is the largest integer that is smaller than *x*.
- `ceiling`: This conversion is an "upwards" truncation. Its value is the smallest integer that is larger than *x*.
- `round`: This conversion is a "closest" truncation. Its value is the integer closest to *x*.

```
? (floor 10.4)
= 10
? (floor -10.4)
= -11
? (ceiling 10.4)
= 11
? (ceiling -10.4)
= -10
? (round 10.4)
= 10
? (round -10.4)
= -10
? (round 10.5)
= 11
? (round -10.5)
= -11
```

`(float x)` [*generic function*]

Summary Conversion to a floating-point number.

Arguments x `<number>`

Result `<float>`

Allocation `<float>` if x is not a float

If the argument x is a floating-point number, the generic function **`float`** returns it as its value. If x is an integer, it is converted into a float, and this value is returned. If x is a ratio, the quotient is computed as a floating-point number and returned.

```
? (float 5.)
= 5.
? (float 5)
= 5.
? (float 1000000000000000000000001)
= 1e+24
? (float 10/3)
= 3.33333333333
```

See also `floatp` (??) `defextern` (??)

0.1.2 Basic operations and comparisons

`(+ &rest numbers)` [*function*]

`(* &rest numbers)` [*function*]

Summary Generic arithmetic operations.

Arguments *numbers* *list of* `<number>`

Result `<number>`

Allocation `<float>, <ratio>` and `<big-int>` if needed

`(/ x y)` [*function*]

Summary Generic arithmetic operations.

Arguments x `<number>`
y `<number>`

Result `<number>`

Allocation `<float>, <ratio>` and `<big-int>` if needed

`(- x &optional y)` [*function*]

Summary Generic arithmetic operations.

Arguments x `<number>`
y `<number>` *Default*

Result `<number>`

Allocation `<float>, <ratio>` and `<big-int>` if needed

These functions perform the basic arithmetic operations.

The functions `+` and `*` are n-ary and are defined for zero arguments; with no arguments, they return their neutral element, which is 0 for `+` and 1 for `*`.

For `+`, `*` and `-`, the type of the result depends on the types of the arguments according to a rule which is an extension of the so-called *floating-point contagion*:

- If at least one argument is a floating-point number, all the arguments are converted to floating-point numbers using `float`; a floating-point specific operation is applied, and the result is a floating-point number.
- If the arguments are integers and ratios, the integers are converted to ratios; a ratio-specific operation is applied, and the result is a ratio.
- If all the arguments are integers, an integer-specific operation is applied, and the result is an integer. Whether this integer is an int or a big-int depends on the range of the result.

The operation `/` follows the same rule, except that the quotient of two integers returns an integer if the division is exact, and a ratio otherwise.

```
? (+ 2 3 4)
= 9
? (+ 2 3/4 4)
= 27/4
? (+ 2 3/4 4.1)
= 6.85
? (+ 3/4 1/4)
= 1/1
? (* 123456789123456789123456789 123456789123456789123456789)
= 15241578780673678546105778281054720515622620750190521
? (* 2/3 3)
= 2/1
? (* 2/3 3.)
= 2.
? (/ 2 3.)
= 0.666666666667
? (/ 4 2)
= 2
? (/ 5 2)
= 5/2
? (- 2 3)
= -1
? (- 7.5)
= -7.5
```

See also `i+` (??) `f+` (??) `i-` (??) `f-` (??) `i*` (??) `f*` (??) `i/` (??) `f/` (??)

(`1+` *number*) [*function*]

(`1-` *number*) [*function*]

Summary Generic arithmetic operations.

Arguments *number* `<number>`

Result `<number>`

Allocation `<float>`, `<ratio>` and `<big-int>` if needed

These functions are shorthands for, respectively, `(+` x `1)` and `(-` x `1)`.

See also `i1+` (??) `i1-` (??)

(`=` *x y* `&rest` *numbers*) [*function*]

(`>` *x y* `&rest` *numbers*) [*function*]

(`>=` *x y* `&rest` *numbers*) [*function*]

(`<` *x y* `&rest` *numbers*) [*function*]

(`<=` *x y* `&rest` *numbers*) [*function*]

Summary Numeric comparisons.

Arguments *x* `<number>`
y `<number>`
numbers *list of* `<number>`

Result `<number> or <null>`

Allocation `<float>, <ratio> and <big-int>` if needed

(`<>` *x y*) [*function*]

Summary Numeric comparison.

Arguments *x* `<number>`
y `<number>`

Result `<number> or <null>`

Allocation `<float>, <ratio> and <big-int>` if needed

These functions perform the usual comparison operations. They return their first argument when the comparison is true, and `()` otherwise. Contagion rules are applied as for the arithmetic operations before actually comparing the numbers, which explains why these functions can allocate.

```
? (= 12 12)
= 12
? (= 12 12 12)
= 12
? (= 12 13)
= ()
? (= 12 12 13)
= ()
? (<> 12 13)
= 12
? (= 5/2 2.5)
5/2
? (> 5/2 2.4)
= 5/2
? (> 5/2 2.4 1)
= 5/2
? (> 5/2 2.4 100)
= ()
? (> 1.2 1.1)
= 1.2
? (> 1.1 1.2)
= ()
? (> 5/3 4/4)
= 5/3
```

See also `i=` (??) `f=` (??) `i>` (??) `f>` (??) `i>=` (??) `f>=` (??) `i<` (??) `f<` (??) `i<=` (??) `f<=` (??)

(`min` *x* `&rest` *numbers*) [*function*]

(`max` *x* `&rest` *numbers*) [*function*]

Summary Maximum and minimum operations.

Arguments *x* `<number>`
numbers *list of* `<number>`

Result `<number>`

Allocation `<float>`, `<ratio>` and `<big-int>` if needed

These functions return respectively the biggest and the smallest of their arguments. These function uses `<` for pairwise comparison of its arguments, which explains why they can allocate.

```
? (min 1 2.4 3/4)
= 3/4
? (max 1 2.4 3/4)
= 2.4
```

See also `imin` (??) `fmin` (??) `imax` (??) `fmax` (??)

(`abs` *x*) [*generic function*]

Summary Absolute value.

Arguments *x* `<number>`

Result `<number>`

Allocation `<float>` and `<big-int>` when required

This function returns the absolute value of x — that is, x itself if it is positive, or minus x if it is negative.

See also `iabs` (??) `fabs` (??)

(`%` *x* *y*) [*function*]

Summary Remainder operation.

Arguments *x* `<integer>`
y `<integer>`

Result `<integer>`

Allocation `<big-int>` if needed

This function returns the remainder of the integer division of x by y. It is always true that `(+ (* b (integer (/ a b))) (% a b))` is equal to `a` if `b` is not zero.

See also `i%` (??)

(`gcd` *x* `&rest` *integers*) [*function*] `(lcm x &rest integers)` [*function*]

Summary GCD and LCM operations.

Arguments *x* `<integer>`
integers *list of* `<integer>`

Result `<integer>`

Allocation `<big-int>` if needed

These functions return respectively the greatest common divisor and the least common multiplier of their arguments.

`from:` [*for iterator*]

`downfrom:` [*for iterator*]

Summary Loops over a numeric range.

The `from:` `for` iterator assigns its loop iteration variable to an ascending series of numeric values. The initial value can be specified by the modifier `from:`. The maximum inclusive value can be specified by the modifier `to:`, or a maximum exclusive value can be specified by the modifier `below:`. Both `to:` and `below:` values cannot be specified together. The step between values can be specified by the modifier `by:`. By default, this iterator uses integer arithmetic. However, by using the `type:` modifier, floating point or generic arithmetic can also be specified. The legal values for the `type:` modifier are the symbols `<integer>`, `<float>`, or `<number>` — in other words, the maximum superclass of the value of the iteration variable. The type is an unevaluated symbol.

The `downfrom:` iterator is like the `from:` iterator, but specifies a descending range. The modifiers which can be used are `to:`, `by:`, `type:`, and `above:`. The modifier `to:` specifies a minimum inclusive final value, and `above:` specifies a minimum exclusive final value. Both cannot be used together. The other modifiers have the same meaning as they do in `from:`. The `by:` value should be positive in both cases; in `from:` it is added to the current value at each iteration, while in `downfrom:` it is subtracted from the current value at each iteration.

```
? (for ((i from: 1 to: 4))
    (printf "%d x %d = %d\n" i i (i* i i)))
1 x 1 = 1
2 x 2 = 4
3 x 3 = 9
4 x 4 = 16
= ()
? (for ((i downfrom: 10.0 by: 2.0 type: <float>)
        (x in: '(a b c d)))
    (collect x)
    (collect i))
= (a 10.0 b 8.0 c 6.0 d 4.0)
```

See also `for` (??)

`collect-sum` [*for gatherer*]

`collect-product` [*for gatherer*]

`collect-max` [*for gatherer*]

`collect-min` [*for gatherer*]

Summary Gathers values into an integer.

These `for` gatherers collect their values together into a single integer. The operation used to coalesce the values depends on the gatherer:

- `collect-sum` — Uses int addition, and starts at 0.
- `collect-product` — Uses int multiplication, and starts at 1.
- `collect-max` — The maximum of the current value and the gatherer value is preserved. The initial value is the value of `(smallest-int)`.
- `collect-min` — The minimum of the current value and the gatherer value is preserved. The initial value is the value of `(biggest-int)`.

```
(defun sum-of-squares (l)
  ;; Returns the sum of the squares of the integer elements of l
  (for ((elt in: l))
    (when (integerp elt)
      (collect-sum elt))))

? (sum-of-squares '(2 3 4))
= 29
? (sum-of-squares '(2 b 3 a 4))
= 29
```