12. Input and output (I/O) of intervals

12.1. Overview. This clause of the standard specifies conversion from a text string that holds an interval literal to an interval internal to a program (input), and the reverse (output). The methods by which strings are read from, or written to, a character stream are language- or implementation-defined, as are variations in some locales (such as specific character case matching).

Containment is preserved on input and output so that, when a program computes an enclosure of some quantity given an enclosure of the data, it can ensure this holds all the way from text data to text results.

In addition to the above I/O, which may incur rounding errors on output and/or input, each interval type $\mathbb{T}$ has an *exact text representation*, via operations that convert any internal $\mathbb{T}$-interval $\boldsymbol{x}$ to a string $\boldsymbol{s}$, and back again to recover $\boldsymbol{x}$ exactly.

12.2. Input. Input is provided for each supported bare or decorated interval type $\mathbb{T}$ by the $\mathbb{T}$-version of `textToInterval`($\boldsymbol{s}$), where $\boldsymbol{s}$ is a string, as specified in §11.11.9. It accepts an arbitrary interval literal $\boldsymbol{s}$ and returns a $\mathbb{T}$-interval enclosing the Level 1 value of $\boldsymbol{s}$.

For 754-conforming types $\mathbb{T}$ the required tightness is specified in §11.11.9. For other types the tightness is implementation-defined.

[*Note. This provides the basis for free-format input of interval literals from a text stream, as might be provided by overloading the >> operator in C++.*]

12.3. Output. An implementation shall provide an operation

`intervalToText`($\boldsymbol{X}$, *cs*)

where *cs* is optional. $\boldsymbol{X}$ is a bare or decorated interval datum of any supported interval type $\mathbb{T}$, and *cs* is a string, the conversion specifier. The operation converts $\boldsymbol{X}$ to a valid interval literal string $\boldsymbol{s}$, see§11.11.1, which shall be related to $\boldsymbol{X}$ as follows, where $\boldsymbol{Y}$ is the Level 1 value of $\boldsymbol{s}$.

(i) Let $\mathbb{T}$ be a bare type. Then $\boldsymbol{Y}$ shall contain $\boldsymbol{X}$, and shall be empty if $\boldsymbol{X}$ is empty.

(ii) Let $\mathbb{T}$ be a decorated type. If $\boldsymbol{X}$ is NaI then $\boldsymbol{Y}$ shall be NaI. Otherwise, write $\boldsymbol{X} = \boldsymbol{x}_{dx}$, $\boldsymbol{Y} = \boldsymbol{y}_{dy}$. Then
- $\boldsymbol{y}$ shall contain $\boldsymbol{x}$, and shall be empty if $\boldsymbol{x}$ is empty.
- dy shall equal dx, except in the case that $dx =$ `com` and overflow occurred, that is, $\boldsymbol{x}$ is bounded and $\boldsymbol{y}$ is unbounded. Then dy shall equal `dac`.

[*Note.* $\boldsymbol{Y}$ *being a Level 1 value is significant. E.g., for a bare type* $\mathbb{T}$*, it is not allowed to convert* $\boldsymbol{X} = \emptyset$ *to the string* `garbage`*, even though converting* `garbage` *back to a bare interval at Level 2 by* $\mathbb{T}$`-textToInterval` *gives* $\emptyset$*, because* `garbage` *has no Level 1 value as a bare interval literal.*]

The tightness of enclosure of $\boldsymbol{X}$ by $\boldsymbol{Y}$ is language- or implementation-defined.

If present, *cs* lets the user control the layout of the string $\boldsymbol{s}$ in a language- or implementation-defined way. The implementation shall document the recognized values of *cs* and their effect; other values are called *invalid*.

If *cs* is invalid, or makes an unsatisfiable request for a given input $\boldsymbol{X}$, the output shall still be an interval literal whose value encloses $\boldsymbol{X}$. A language- or implementation-defined extension to interval literal syntax may be used, to make it obvious that this has occurred. [*Example. Suppose, for uncertain form, that* m *is undefined or* r *is "unreasonably large". Then a string such as* `[Entire!uncertain form conversion error]` *might be produced. The implementation of* `textToInterval` *would need to accept this string as meaning the same as* `[Entire]`.]

Among the user-controllable features should be the following, where l, u are the interval bounds for inf-sup form, and m, r are the base point and radius for uncertain form, as defined in §11.11.1.

(i) It should be possible to specify the preferred overall field width (the length of $\boldsymbol{s}$), and whether output is in inf-sup or uncertain form.

(ii) It should be possible to specify how Empty, Entire and NaI are output, e.g., whether lower or upper case, and whether Entire becomes `[Entire]` or `[-Inf, Inf]`.

(iii) For l, u and m, it should be possible to specify the field width, and the number of digits after the point or the number of significant digits. For r, which is a non-negative integer ulp-count, it should be possible to specify the field width. There should be a choice of radix, at least between decimal and hexadecimal.

(iv) For uncertain form, it should be possible to select the default symmetric form, or the one sided (`u` or `d`) forms. It should be possible to choose whether an exponent field is absent (and m is output to a given number of digits after the point) or present (and m is output to a given number of significant digits).

(v) It should be possible to output the bounds of an interval without punctuation, e.g. `1.234  2.345` instead of `[1.234, 2.345]`. For instance this might be a convenient way to write intervals to a file for use by another application.

If *cs* is absent, output should be in a general-purpose layout (analogous, e.g., to the `%g` specifier of `fprintf` in C). There should be a value of *cs* that selects this layout explicitly.
[*Note. This provides the basis for free-format output of intervals to a text stream, as might be provided by overloading the* `<<` *operator in C++.*]

If $\mathbb{T}$ is a 754-conforming bare type, there shall be a value of *cs* that produces behavior identical with that of `intervalToExact`, below. That is, the output is an interval literal that, when read back by $\mathbb{T}$-`textToInterval`, recovers the original datum exactly.

12.4. Exact text representation. For any supported bare interval type $\mathbb{T}$ an implementation shall provide operations `intervalToExact` and `exactToInterval`. Their purpose is to provide a portable exact representation of every bare interval datum as a string.

These operations shall obey the **recovery requirement**:

> For any $\mathbb{T}$-datum $\boldsymbol{x}$, the value $\boldsymbol{s} = \mathbb{T}$-`intervalToExact`$(\boldsymbol{x})$ is a string,
> such that $\boldsymbol{y} = \mathbb{T}$-`exactToInterval`$(\boldsymbol{s})$ is defined and equals $\boldsymbol{x}$.

[*Note. From §11.3, this is equality as datums:* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *have the same Level 1 value and the same type. They may differ at Level 3, e.g., a zero endpoint might be stored as* -0 *in one and* $+0$ *in the other.*]

If $\mathbb{T}$ is a 754-conforming type, the string $\boldsymbol{s}$ shall be an interval literal which, for nonempty $\boldsymbol{x}$, is of inf-sup type, with the lower and upper bounds of $\boldsymbol{x}$ converted as described in §12.4.1. Note that for such $\boldsymbol{s}$, the operation `exactToInterval` is functionally equivalent to `textToInterval`.

If $\mathbb{T}$ is not 754-conforming, there are no restrictions on the form of the string $\boldsymbol{s}$ apart from the above recovery requirement. However, the representation should aim to display the values of the parameters that define the underlying mathematical model, in a human-readable way.

The algorithm by which `intervalToExact` converts $\boldsymbol{x}$ to $\boldsymbol{s}$ is regarded as part of the definition of the type and shall be documented by the implementation.
[*Example. Writing a* `binary64` *floating point datum exactly in hexadecimal-significand form passes the "readability" test since it displays the parameters sign, exponent and significand. Dumping its 64 bits as 16 hex characters does not.*]

Since `exactToInterval` creates an interval from non-interval data, it is a constructor similar to `textToInterval`, and (see §11.11.9), shall return Empty and signal a language- or implementation-defined exception when its input is invalid.

12.4.1. *Conversion of 754 numbers to strings.* A 754 format $\mathbb{F}$ is defined by the parameters: b = the radix, 2 or 10; p = the number of digits in the significand (precision); $emax$ = the maximum exponent; $emin = 1 - emax$ = the minimum exponent (see 754-2008 §3.3).

A finite $\mathbb{F}$-number x can be represented $(-1)^s \times b^e \times m$ where $s = 0$ or 1, e is an integer, $emin \le e \le emax$, and m has a p-digit radix b expansion $d_0.d_1d_2\ldots d_{p-1}$, where d_i is an integer digit $0 \le d_i < b$ (so $0 \le m < b$). As used within interval literals, x denotes a real number, with no distinction between -0 and $+0$. To make the representation unique, constraints are imposed in three mutually exclusive cases:

– A *normal* number, with $|x| \ge b^{emin}$, shall have $d_0 \ge 1$ (so $1 \le m < b$).
– A *subnormal* number, with $0 < |x| < b^{emin}$, shall have $e = emin$, which implies $d_0 = 0$ (so $0 < m < 1$).
– *Zero*, $x = 0$, shall have sign bit $s = 0$ and exponent $e = 0$ (and necessarily $m = 0$).

[*Note. For* $b = 2$ *the standard form used by 754 is the same as this, except for replacing zero by two signed zeros, with exponent* $e = emin$*. For* $b = 10$*, there is also the difference that 754 normal numbers have several representations if they need fewer than* p *digits in their expansion. The standard form above chooses the representation with smallest quantum, which is the unique one having* $d_0 \ne 0$*.*]

The rules given below for converting x to a string *xstr* allow user-, language- or implementation-defined choice while ensuring the values of s, m and e are easily found from *xstr* in each of these cases, even without knowledge of the format parameters p, *emax*, *emin*.

xstr is the concatenation of: a sign part *sstr*; a significand part *mstr*; and an exponent part *estr*. If $b = 2$, the hex-indicator `"0x"` is prefixed to *mstr*.

sstr is `"-"` or an optional `"+"`, as appropriate.

If $b = 10$, *mstr* is the (decimal) expansion $d_0.d_1d_2 \dots d_{p-1}$, optionally abbreviated by removing some or all trailing zeros. If this leaves no digits after the point, the point may be removed. If $b = 2$, *mstr* is formed from the (binary) expansion $d_0.d_1d_2 \dots d_{p-1}$, abbreviated in the same way, and then converted to a hexadecimal string $D_0.D_1 \dots$ (so necessarily D_0 is 1 if x is normal, 0 if x is subnormal or zero).

estr consists of `"e"` if $b = 10$, `"p"` if $b = 2$, followed by the exponent e written as a signed decimal integer, with the sign optional if $e \geq 0$.

[*Examples. In any binary format, the number* 2 *(with* $s = 0$, $m = 1$, $e = 1$*) may be written as* `0x1p1` *or* `+0x1.0p+01`*, etc., but not as* `0x2p0`*; while* $\frac{1}{2}$ *may be written as* `0x1p-1` *or* `+0x1.0p-01`*, etc. The number* -4095 *(with* $s = 1$, $m = \frac{4095}{2048}$, $e = 11$*) may be written as* `-0x1.ffep+11`.

In `decimal32` *(see 754-2008 Table 3.6), which has* $p = 7$*, the smallest positive normal number may be written* `1e-95` *or* `+1.000000e-95`*, etc.; and the next number below it as* `0.999999e-95`*. The smallest positive number can be written* `0.000001e-95`.]

Above, alphabetic characters have been written in lowercase, but may be in either case.

A shorter form for subnormal numbers may be used, normalized by requiring $d_1 \neq 0$; however, to find the canonical m and e from *xstr* one then needs to know *emin*. For instance the smallest positive `decimal32` number $x =$ `0.000001e-95` has the shorter form `0.1e-101`, but to deduce that x has $m = 0.000001$ and $e = -95$ one needs to know that $emin = -95$ for this format.

12.4.2. *Exact representations of comparable types.* The exact text representation of a bare interval of any type should also be a valid exact representation in any wider (in the sense of §11.5.1) type, which when converted back produces the mathematically same interval.

That is, let type $\mathbb{T}'$ be wider than type $\mathbb{T}$. Let $\boldsymbol{x}$ be a $\mathbb{T}$-interval and let

$$\boldsymbol{s} = \mathbb{T}\texttt{-intervalToExact}(\boldsymbol{x}).$$

Then

$$\boldsymbol{x}' = \mathbb{T}'\texttt{-exactToInterval}(\boldsymbol{s})$$

should be defined and equal to $\mathbb{T}'$-`convertType`(x).

[*Note. If* $\mathbb{T}$ *and* $\mathbb{T}'$ *are 754-conforming types, this property holds automatically, because of the properties of* `textToInterval` *and the fact that* $\boldsymbol{s}$ *is an interval literal.*]