`nextOut`() relaxes the requirement for correct (rather than, say, faithful) rounding, which might be hard to achieve for some special functions at some arguments.

NOTE 3—The input box $\boldsymbol{x}$ might have components of a different type from the result type $\mathbb{T}$, in the case of 754-conforming mixed-type operations 12.6.2. The $\text{hull}_{\mathbb{T}}$ in (28) forces these to have type $\mathbb{T}$, so each component of $\boldsymbol{x}$ is widened by at least the local spacing of $\mathbb{F}$-numbers, at each finite bound.
[Example. Let $\mathbb{T}$ be a 2-digit decimal inf-sup type. Then `nextOut` *widens the $\mathbb{T}$-interval $\boldsymbol{x} = [2.4, 3.7]$ to $[2.3, 3.8]$—an ulp at each end, see* **??**. *But an operation might accept 4-digit decimal inf-sup inputs, and $\boldsymbol{x}$ might be $[2.401, 3.699]$. Then* `nextOut`$(\boldsymbol{x})$ *is* $[\texttt{nextDown}(2.401), \texttt{nextUp}(3.699)] = [2.4, 3.7]$*, giving an insignificant widening. But*

$$\texttt{nextOut}(\text{hull}_{\mathbb{T}}([2.401, 3.699])) = \texttt{nextOut}([2.4, 3.7]) = [2.3, 3.8]$$

gives a widening comparable with the precision of $\mathbb{T}$.]

12.10.2. Accuracy requirements

Following the categories of functions in Table 9.1, the accuracy of the *basic operations*, the *integer functions* and the *absmax functions* shall be *tightest*. The accuracy of the *cancellative addition and subtraction* operations of 10.5.6 is specified in 12.12.5.

For all other operations in Table 9.1, for the reverse mode operations of Table 10.1, and for the recommended operations of Table 10.5 and Table 10.6, the accuracy shall be *valid*, and, for inf-sup types, should be *accurate*. For any operation in these four tables, if any input is Empty the result shall be Empty.

12.10.3. Documentation requirements

An implementation shall document the tightness of each of its interval operations for each supported bare interval type. This shall be done by dividing the set of possible inputs into disjoint subsets ("ranges") and stating a tightness achieved in each range. This information may be supplemented by further detail, e.g., to give accuracy data in a more appropriate way for a non-inf-sup type.

[Example. Sample tightness information for the sin *function might be*

Operation	*Type*	*Tightness*	*Range*
sin	`infsup binary64`	*tightest*	*for any* $\boldsymbol{x} \subseteq [-10^{15}, 10^{15}]$
		accurate	*for all other* $\boldsymbol{x}$.

]

Each operation should be identified by a language- or implementation-defined name of the Level 1 operation (which might differ from that used in this standard), its output type, its input type(s) if necessary, and any other information needed to resolve ambiguity.

12.11. Interval and number literals

12.11.1. Overview

This subclause extends the specifications of 9.4 to define interval literals in the set-based flavor. Interval literals are used as input to `textToInterval` in 12.12.7, and in Clause 13, Input/Output. The following definitions and usages from 9.4 are unchanged: integer literal; value of a literal; valid and invalid string; case insensitivity; unit in last place; the possibility of implementation-defined or locale-dependent variations and the requirement to document them.

12.11.2. Number literals

In addition to those specified in 9.4, the following forms of number literal shall be provided.

d) Either of the strings `inf` or `infinity` optionally preceded by a plus sign, with value $+\infty$; or preceded by a minus sign, with value $-\infty$.

TABLE 12.1. Portable interval literal examples.

Form	Literal	Exact decorated value
Special	`[ ]`	$\text{Empty}_{\texttt{trv}}$
	`[entire]`	$[-\infty, +\infty]_{\texttt{dac}}$
Inf-sup	`[1.e-3, 1.1e-3]`	$[0.001, 0.0011]_{\texttt{com}}$
	`[-Inf, 2/3]`	$[-\infty, 2/3]_{\texttt{dac}}$
	`[0x1.3p-1,]`	$[19/32, +\infty]_{\texttt{dac}}$
	`[,]`	$\text{Entire}_{\texttt{dac}}$
Uncertain	`3.56?1`	$[3.55, 3.57]_{\texttt{com}}$
	`3.56?1e2`	$[355, 357]_{\texttt{com}}$
	`3.560?2`	$[3.558, 3.562]_{\texttt{com}}$
	`3.56?`	$[3.555, 3.565]_{\texttt{com}}$
	`3.560?2u`	$[3.560, 3.562]_{\texttt{com}}$
	`-10?`	$[-10.5, -9.5]_{\texttt{com}}$
	`-10?u`	$[-10.0, -9.5]_{\texttt{com}}$
	`-10?12`	$[-22.0, 2.0]_{\texttt{com}}$
	`-10??u`	$[-10.0, +\infty]_{\texttt{dac}}$
	`-10??`	$[-\infty, +\infty]_{\texttt{dac}}$
NaI	`[nai]`	$\text{Empty}_{\texttt{ill}}$
Decorated	`3.56?1_def`	$[3.55, 3.57]_{\texttt{def}}$

12.11.3. Bare intervals

In addition to those specified in 9.4, the following forms of bare interval literal shall be supported. (A) marks that a new form of literal is Added; (C) marks an existing form, Changed by adding an extra feature.

a) Inf-sup form (C): In the string `[` l `,` u `]`, the bound l may be $-\infty$ and u may be $+\infty$. Any of l and u may be omitted, with implied values $l = -\infty$ and $u = +\infty$, respectively, e.g., `[,]` denotes Entire.

b) Uncertain form (C): This form, with radius empty or *ulp-count*, is adequate for narrow (hence bounded) intervals, but is severely restricted otherwise. Uncertain form with radius `?` is used for unbounded intervals, e.g., m`??d` denotes $[-\infty, m]$, m`??u` denotes $[m, +\infty]$ and m`??` denotes Entire with m being like a comment.

c) Special values (A): The strings `[ ]` and `[ empty ]`, whose bare value is Empty; the string `[ entire ]`, whose bare value is Entire. Here and below, space shown between elements of a literal is optional: it denotes zero or more space characters. E.g., one may write `[empty]` or `[ empty ]`, etc.

12.11.4. Decorated intervals

The following forms of decorated interval literal shall be supported.

a) A bare interval literal $\boldsymbol{sx}$.

   If $\boldsymbol{sx}$ has the bare value $\boldsymbol{x}$, then $\boldsymbol{sx}$ has the value `newDec`$(\boldsymbol{x})$, see 11.5. Otherwise $\boldsymbol{sx}$ has no value as a decorated interval literal.

b) A bare interval literal $\boldsymbol{sx}$, an underscore "`_`", and a 3-character decoration string $\boldsymbol{sd}$; where $\boldsymbol{sd}$ is one of `trv`, `def`, `dac` or `com`, denoting the corresponding decoration dx.

   If $\boldsymbol{sx}$ has the bare value $\boldsymbol{x}$, and if $\boldsymbol{x}_{dx}$ is a permitted combination according to 11.4, then $\boldsymbol{sx\_sd}$ has the value $\boldsymbol{x}_{dx}$. Otherwise $\boldsymbol{sx\_sd}$ has no value as a decorated interval literal.

c) The string `[ nai ]`, with the value $\text{Empty}_{\texttt{ill}}$.

[Examples. Table 12.1 illustrates valid portable interval literals. These strings are not valid portable interval literals: `empty`, `[5?1]`, `[1_000_000]`, `[ganz]`, `[entire!comment]`, `[inf]`, `5???u`, `[nai]_ill`, `[]_ill`, `[]_def`, `[0,inf]_com`*.]*

TABLE 12.2. Differences from the general grammar for literals in Table 9.3. In the left column, 'A' marks a new term that is Added, and 'C' marks a term that is Changed from its definition in Table 9.3. The change is always in the form of added alternative(s) on the right hand side—these are underlined.

A	infNumLit	{sign}? ("inf" | "infinity")
C	numberLiteral	{decNumLit} | {hexNumLit} | {ratNumLit} | {infNumLit}
C	radius	{natural} | "?"
A	emptyIntvl	"[" {sp} "]" | "[" {sp} "empty" {sp} "]"
A	entireIntvl	"[" {sp} "entire" {sp} "]"
A	specialIntvl	{emptyIntvl} | {entireIntvl}
C	bareIntvlLiteral	{pointIntvl} | {infSupIntvl} | {uncertIntvl} | {specialIntvl}
A	NaI	"[" {sp} "nai" {sp} "]"
C	decorationLit	"trv" | "def" | "dac" | "com"
C	intervalLiteral	{bareIntvlLiteral} | {bareIntvlLiteral} "_" {decorationLit} | {NaI}

12.11.5. Grammar for portable literals

The syntax of portable integer and number literals and of portable bare and decorated interval literals in this flavor is defined by integerLiteral, numberLiteral, bareIntvlLiteral and intervalLiteral, respectively, in a variant of the grammar defined in Table 9.3. The differences are shown in Table 12.2.

The constructor `textToInterval` (12.12.7, 13.2) of any implementation shall accept any portable interval. An implementation may restrict support of some input strings (too long strings or strings with a rational number literal). Nevertheless, the constructor shall always return a Level 2 interval (possibly Entire in this case) that contains the Level 1 interval.

An implementation may support interval literals of more general syntax (for example, with underscores in significand). In this case there shall be a value of conversion specifier *cs* that restricts output strings of `intervalToText` 13.3 to the portable syntax.

12.12. Required operations on bare and decorated intervals

An implementation shall provide a $\mathbb{T}$-version, see 12.9, of each operation listed in 12.12.1 to 12.12.10, for each supported type $\mathbb{T}$. That is, those of the $\mathbb{T}$-version's inputs and outputs that are intervals are of type $\mathbb{T}$ (or the corresponding decorated type), except for the conversion operation of 12.12.10 whose output is of type $\mathbb{T}$ and whose input is of any type.

The implementation shall provide the type-independent decoration operations of 12.12.11. It shall provide the reduction operations of 12.12.12 for the parent formats of supported 754-conforming types.

Operations in this subclause are described as functions with zero or more input arguments and one return value. It is language- or implementation-defined whether they are implemented in this way: for instance, two-output division, described in 12.12.3 as a function returning an ordered pair of intervals, might be implemented as a procedure `mulRevToPair`$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}_1, \boldsymbol{z}_2)$ with input arguments $\boldsymbol{x}$ and $\boldsymbol{y}$ and output arguments $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$.

An implementation, or a part thereof, that is 754-conforming shall provide mixed-type operations, as specified in 12.6.2, for the following operations, which correspond to those that IEEE Std 754-2008 requires to be provided as *formatOf* operations.

`add, sub, mul, div, recip, sqrt, sqr, fma`.

An implementation may provide more than one version of some operations for a given type. For instance, it may provide an "accurate" version in addition to a required "tightest" one, to offer a trade-off of accuracy versus speed or code size. How such a facility is provided is language- or implementation-defined.