Interval Arithmetic Specification

Dmitri Chiriaev and G. William Walster

Revised March 13, 2000

Abstract

Compiler support for interval arithmetic requires a specification of both the syntax and semantics of the implementation. The Fortran 95 specification contained herein defines a set of extended real intervals and their internal representation for IEEE 754 compliant processors. The defined set of extended real intervals is closed with respect to arithmetic operations and interval enclosures of real expressions. For mathematical details, see [20] and [23].

Contents

1 Introduction

This specification defines a set of representable extended real intervals, a subset of supported intervals and their internal representation for IEEE 754 compliant processors. It is also a specification of the needed additions to the syntax and semantics of Fortran 95 with which to implement, test and document support for interval data types. It is not a complete implementation specification, although some important implementation issues are addressed.

Real interval arithmetic operations are interval enclosures of their point counterparts. So are rational expressions[1] computed using interval arithmetic. This fact, known as the fundamental theorem of interval arithmetic, is important because an interval enclosure of a real expression contains the expression's range over the sub-domain defined by its interval arguments [12]. The property of including the range of a expression in an interval result is also referred to as "containment".

Sendov [16] extended the domain of interval arithmetic to include unbounded intervals. Walster et al [18] defined the set of values that interval enclosures of extended point expressions must contain. Walster [20] defined a closed system of extended real intervals that includes enclosures of intrinsics and relations where point results are undefined. Walster and Hansen [23] extended the fundamental theorem to include multi–valued and irrational functions. The practical consequences of these results as implemented herein are:

[1] The term "expression" is used to include both single and multi–valued functions.

1. Simple representation: Each supported interval can be internally represented using a pair of IEEE floating-point numbers.

2. Algorithm efficiency: Branching is minimized and there are opportunities for additional hardware support.

3. Closure: All arithmetic operations on any supported intervals produce supported intervals.

4. Sharp Results: Whenever possible, sharp[2] interval results are produced.

5. Simplicity: User interface complexity is minimized.

A complete set of intrinsic interval enclosures of point intrinsics and operations for Fortran is defined using [7] together with [18], [20] and [23]. For IEEE 754 compliant processors this collection of definitions yields a consistent closed set of interval operations and intrinsics, even in the presence of arguments and outcomes that would otherwise raise IEEE exceptions. Appendix A contains algorithms for interval operations `+`, `-`, `*`, and `/` on IEEE 754 compliant processors.

This specification is written to be consistent as practical with the Fortran 95 Standard. Additions to this specification for future versions of Fortran and other languages will be provided in future versions.

2 Interval Implementation Quality

Speed and sharpness define the quality of an interval implementation. The result of an interval operation is *sharp* if the resulting width, $w([a,b]) = b - a$, is as small as possible without violating the containment constraint.

The two components of interval implementation quality do not necessarily conflict. Taking more time to compute sharp elementary and intermediate results can cause an interval algorithm to execute faster.

Without special interval hardware, speed forces a pair of IEEE 754 floating–point numbers to be used when representing the infimum and supremum of an interval. Speed can also be achieved by eliminating exceptional events. This requires an interval system that is *closed* with respect to arithmetic operations, including division by an interval containing zero and the evaluation of expressions including both relations and functions.

[2] An interval X is "sharper" than interval Y, if `WID(X)` < `WID(Y)` (see section 16.8), that is, assuming both X and Y contain the correct answer.

Sharpness of interval results can be improved by distinguishing both between zero and underflow, on the one hand, and between infinity and overflow, on the other. Sharpness of interval results can also be improved by permitting the result of interval operations to be represented by a set of disjoint intervals.

Quickly computing sharp interval results can be achieved with various amounts of hardware support. In some cases, different internal representations are required. Nevertheless, binary file compatibility can be maintained, see section 21.13.

2.1 The "Simple", "Sharp", and "Full" Interval Systems

The three interval systems, "Simple", "Sharp", and "Full", are designed to introduce increasing result sharpness at the cost of implementation complexity and hardware support. All three systems use extended real interval arithmetic as their base. The "Sharp" system adds the ability to distinguish between underflow and zero, on the one hand, and between overflow and infinity, on the other. This makes it possible to preserve sign information which is otherwise lost. The "Full" system adds the ability to sharply represent division by intervals containing zero as the union of two semi-infinite intervals. All three systems are closed with respect to interval arithmetic operations. The present specification implements the "Simple" system and requiring no hardware support other than IEEE 754 compliance.

3 Extended Real Intervals

Let $\mathbb{R}$ denote the set of real numbers: $\mathbb{R} \equiv \{x \mid -\infty < x < +\infty\}$. A real interval, or just an interval, $X = [\underline{x}, \overline{x}]$, is a closed, bounded subset of the real numbers, $\mathbb{R}$:

$$X \equiv [\underline{x}, \overline{x}] \equiv \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \quad ,$$

where $\underline{x}$ and $\overline{x}$ denote the *left endpoint* or infimum[3] and the *right endpoint* or supremum[4] of the interval X. The set of real intervals is denoted by $I\mathbb{R}$. An interval is a *point* or *degenerate* interval if $X = [x, x]$ or $\underline{x} = \overline{x}$. The infimum of an interval is always less than or equal to the supremum.

The set of extended real numbers, $\mathbb{R}^*$, is the set of real numbers, $\mathbb{R}$, extended with the two ideal points plus and minus infinity: $\mathbb{R}^* = \mathbb{R} \cup \{-\infty\} \cup \{+\infty\}$. The set of extended real intervals, $I\mathbb{R}^*$, is the set of real intervals, $I\mathbb{R}$, extended with the empty interval, $\emptyset$, and

[3] The largest number that is less than or equal to each of a given set of real numbers.

[4] The smallest number that is greater than or equal to each of a given set of real numbers.

intervals with one or both infinite endpoints.

$$I\!I\!R^* \equiv \begin{cases} I\!I\!R \cup \\ \emptyset \cup \\ \{[-\infty, x] \mid x \in I\!R\} \cup \\ \{[x, +\infty] \mid x \in I\!R\} \cup \\ [-\infty, -\infty] \cup \\ [-\infty, +\infty] \cup \\ [+\infty, +\infty] \end{cases}$$

where intervals with infinite endpoints are interpreted as follows:

$$\begin{aligned} [-\infty, x] &\equiv \{z \in I\!R^* \mid z \leq x,\ x \in I\!R\ \} \\ [x, +\infty] &\equiv \{z \in I\!R^* \mid z \geq x,\ x \in I\!R\ \} \\ [-\infty, -\infty] &\equiv \{z \in I\!R^* \mid z < x, \forall x \in I\!R\} \\ [-\infty, +\infty] &\equiv I\!R^* \\ [+\infty, +\infty] &\equiv \{z \in I\!R^* \mid z > x, \forall x \in I\!R\} \end{aligned}$$

The set of values that an arithmetic operation on extended intervals must contain is the "containment–set" of the operation, is denoted $\{X\ op\ Y\}$, and as conceived my Moore is defined :

Definition 1

$$\begin{aligned} \{X\ op\ Y\} &= \{\xi \mid \xi = x\ op\ y,\ x \in X\ and\ y \in Y\}\ ;\ where \\ op &= \{+, -, \times, \div\}\ ,\ X \in I\!I\!R\ ,\ Y \in I\!I\!R \end{aligned}$$

and in the case of division, $0 \notin Y$*; because* $x \div y$ *is undefined if* $y = 0$*.*

Remark 1 *In case* $x\ op\ y$ *is not uniquely defined,* $X\ op\ Y$ *must still contain the set of all possible values of* ξ *to which* $\xi = x\ op\ y$ *can be equal. This set is the containment set of the expression* $\xi = x\ op\ y$ *and is denoted* $\{[x\ op\ y]\}$*, see [18].*

Remark 2 *An interval operation produces an interval containing the set of all results obtainable by performing the operation in question on every element of the argument intervals. Therefore* $X\ op\ Y$ *must contain* $\{X\ op\ Y\}$*.*

Neither the real nor the extended real number systems are closed with respect to arithmetic operations. In the real number system, division by zero is undefined. In the extended real

number system, neither division by zero nor the following indeterminate forms are defined:

$(+\infty) - (+\infty)$	$(+\infty) \times 0$	$(+\infty) \div (+\infty)$
$(-\infty) + (+\infty)$	$(-\infty) \times 0$	$(-\infty) \div (+\infty)$
$(+\infty) + (-\infty)$	$0 \times (+\infty)$	$(+\infty) \div (-\infty)$
$(-\infty) - (-\infty)$	$0 \times (-\infty)$	$(-\infty) \div (-\infty)$

(1)

The following notation makes explicit the identities between the set and interval representations, both for the point at 0 and for the ideal points at $\pm\infty$:

$$\begin{aligned} [-\infty, -\infty] &\equiv -\infty \ , \\ [+\infty, +\infty] &\equiv +\infty \ , \\ [0, 0] &\equiv 0. \end{aligned}$$

The relationship between projective infinity, $\{\pm\infty\}$, and affine infinities, $-\infty$ and $+\infty$, is:

$$-\infty \cup +\infty \equiv \{\pm\infty\} \tag{2}$$

The following reciprocity relations exist between zero and projective infinity, which can then be related to affine infinities using (2):

$$\begin{aligned} [0, 0] &= -[0, 0] \ ; \\ \{\pm\infty\} &= -\{\pm\infty\} \ ; \\ 1/(-\infty) &= 1/(+\infty) = 1/\{\pm\infty\} = [0, 0] \ ; \text{ and,} \\ 1/[0, 0] &\equiv \{\pm\infty\} \ . \end{aligned}$$

To close the system of extended intervals with respect to the arithmetic operators, it is necessary to define the result of interval operations on operands involving $\emptyset$, as well as indeterminate forms, including those in table 1. This has been done in [18]. Definition 1 does not apply to extended intervals or division by zero or the indeterminate forms in (1). If $x \ op \ y$ is indeterminate, then define $\{[x \ op \ y]\}$ to be the set of all possible values of ξ to which $\xi = x \ op \ y$ can be equal. If $x \ op \ y$ is determinate, then $\{[x \ op \ y]\} = x \ op \ y$. The following required results are derived from topological closure of arithmetic operations using results in [20].

$$\{\emptyset \ op \ X\} = \emptyset \ \text{ for } \ op \ \in \{+, -, \times, \div\} \ \text{ and } \ \forall X \in I\!R^* \ ; \tag{3}$$

$$\{X \ op \ \emptyset\} = \emptyset \ \text{ for } \ op \ \in \{+, -, \times, \div\} \ \text{ and } \ \forall X \in I\!R^* \ ; \tag{4}$$

$$\{[(+\infty) - (+\infty)]\} = I\!R^* \ ; \tag{5}$$

$$\{[(-\infty)+(+\infty)]\} = I\!R^* \ ; \tag{6}$$
$$\{[(+\infty)+(-\infty)]\} = I\!R^* \ ; \tag{7}$$
$$\{[(-\infty)-(-\infty)]\} = I\!R^* \ ; \tag{8}$$
$$\{[0/0]\} = I\!R^* \ ; \tag{9}$$
$$\{[(+\infty)\times 0]\} = I\!R^* \ ; \tag{10}$$
$$\{[(-\infty)\times 0]\} = I\!R^* \ ; \tag{11}$$
$$\{[0\times(+\infty)]\} = I\!R^* \ ; \tag{12}$$
$$\{[0\times(-\infty)]\} = I\!R^* \ ; \tag{13}$$
$$\{[a/0\} = \{\pm\infty]\} \subset I\!R^* \quad \forall\ 0 < a \ ; \tag{14}$$
$$\{[(-a)/0]\} = \{\pm\infty\} \subset I\!R^* \quad \forall\ 0 < a \ ; \tag{15}$$
$$\{[(+\infty)/(+\infty)]\} = [0, +\infty] \ ; \tag{16}$$
$$\{[(-\infty)/(+\infty)]\} = [-\infty, 0] \ ;. \tag{17}$$
$$\{[(+\infty)/(-\infty)]\} = [-\infty, 0] \ ; \text{and}, \tag{18}$$
$$\{[(-\infty)/(-\infty)]\} = [0, +\infty] \ . \tag{19}$$

The result of equations (14) and (15) is unsigned or projective infinity, while not sharp containment is preserved if $I\!R^*$ is returned instead of $\{\pm\infty\}$, because projective infinity is contained in $I\!R^*$.

The indeterminate forms in (1) are undefined in the point system because these forms define multi–valued, not single–valued functions. Because intervals are *sets* of values, the results in (3) can be used to return containing intervals.

The results from the definitions in (3) through (19) can be used to generalize Definition 1:

Definition 2

$$\{X\ op\ Y\} = \{\{[x\ op\ y]\} \mid x \in X \quad and \quad y \in Y\} \ ; \textit{ where}$$

$op = \{+, -, \times, \div\}$, $X \in I\!I\!R^*$, $Y \in I\!I\!R^*$, $\{[x\ op\ y]\} = x\ op\ y$ *when* $x\ op\ y$ *is defined, but whenever* $x\ op\ y$ *is indeterminate, the definitions of* $\{[x\ op\ y]\}$ *in (3) through (19) are used instead.*

4 Internal Representation using IEEE numbers

To support arithmetic operations on the complete set of extended real intervals requires non–IEEE arithmetic operators, or the sacrifice of run–time performance. An alternative is to exclude the intervals $[-\infty, -\infty]$ and $[+\infty, +\infty]$ from the set of supported intervals. As a consequence, the invalid operation exception $(+\infty) - (+\infty)$ is avoided in inlined IEEE

floating–point addition and subtraction operations. Without loss of containment or closure, any containing interval can be returned for the result of an operation on supported intervals that produces an unsupported interval result. In particular, $[-\texttt{inf}, -\overline{\texttt{fp}}]$ can be returned in place of `[-inf,-inf]` and $[+\overline{\texttt{fp}}, +\texttt{inf}]$ can be returned in place of `[+inf]`, where $\overline{fp}$ denotes the largest representable floating–point number[5].

The following design objectives are supported by the chosen mapping of interval endpoints onto the set of IEEE repesentable floating-point numbers:

- represent the empty, $\emptyset$, and entire, $\mathbb{R}^*$, intervals;
- use IEEE features to implement the relative dominance of the empty ($\emptyset$) and the entire ($\mathbb{R}^*$) intervals; and,
- use the properties of IEEE floating-point arithmetic to achieve sharpness and speed.

Justification for limiting the first release of Sun's compiler support for intervals to the "Simple" system is contained in [21]. While the "Simple" system produces results that are less sharp in the presence of underflow and overflow than the "Sharp" and "Full" systems in [20], simplicity is believed to be more important than sharpness in the first interval compiler release.

The empty interval, $\emptyset$, is represented using the unique internal representation $[NaN_\emptyset, NaN_\emptyset]$; where $NaN_\emptyset$ is a non-default quiet not-a-number. To provide efficient hardware support for the "Sharp" and "Simple" systems, $NaN_\emptyset$, must be unique and unobtainable in any other way. The result of operations such as IEEE `0*inf`, `inf/inf`, or `0/0`, produce a default NaN. Using a non default NaN to represent the empty interval enables empty intervals to be efficiently propagated without introducing branches.

The entire interval $\mathbb{R}^*$ is internally represented using $[-\texttt{inf}, +\texttt{inf}]$.

Zero interval endpoints are internally represented using either plus or minus zero. In the "Simple" system, the sign of zero is ignored. Processors that implicitly initialize variables to zero may initialize variables of interval type to `[-0,-0]`, `[+0,+0]`, `[+0,-0]` or `[-0,+0]`.

The following examples for the "Simple" system are obtained form Definition 2 and equations (5) – (19).

Given $a, b, c, d \in \mathbb{R}$ and $a, b, c, d > 0$

```
[a, +inf] * [-c,d] = [-inf,+inf]
[-inf, -b] * [-c,d] = [-inf,+inf]
```

[5] The notation $\overline{fp}$ and -$\overline{fp}$ is not meant to imply that floating–point numbers are symmetrically distributed around zero

```
[a, +inf] * [± 0 , d] = [-inf,+inf]
[-inf, -b] * [± 0,d] = [-inf,+inf]
[-inf, b] * [± 0,d] = [-inf,+inf]
[-inf, b] * [-c,± 0 ] = [-inf,+inf]
```

For all intervals $X \in I\!R^*$:

```
X / [± 0,d] = [-inf,+inf]
X / [c,± 0] = [-inf,+inf]
X / [-c,d] = [-inf,+inf]
[a,+inf] / [c,+inf] = [0,+inf]
[-inf,-b] / [-inf,-d] = [0,+inf]
[-inf,-b] / [c,+inf] = [-inf,0]
[a, +inf] / [-inf,-d] = [-inf,0]
```

The following table shows how (in the "Simple" system) internal binary representations are interpreted on output ("x" stands for an external character representation of x). In this table, x $\neq$ inf and x $\neq$ 0.

Internal binary	Output	External character
[± 0, ± 0]	$\longrightarrow$	[0,0]
[± 0,x]	$\longrightarrow$	[0,"x"]
[x, ± 0]	$\longrightarrow$	["x",0]
[x, +inf]	$\longrightarrow$	["x", +INF]
[+0, +inf]	$\longrightarrow$	[0,+INF]
[-inf,x]	$\longrightarrow$	[-INF,"x"]
[-inf, +0]	$\longrightarrow$	[-INF,0]

The following table shows how (in the "Simple" system) character strings are converted into the internal binary representation ("x" stands for an external character representation of x). In this table, x $\neq$ inf and x $\neq$ 0.

Internal binary	Input	External character
[±0,x]	⟵	[0,"x"]
[±0,x]	⟵	[1e-5000, "x"]
[x,±0]	⟵	["x",0]
[x,±0]	⟵	["x",-1E-5000]
[x,+inf]	⟵	["x",+INF]
[±0,+inf]	⟵	[0, +INF]
[-0,+0]	⟵	[0,0]
[-inf,±0]	⟵	[-INF,0]
[-inf,±0]	⟵	[-INF, -1E-5000]
[-inf,x]	⟵	[-INF,"x"]
[-inf, $-\overline{fp}$]	⟵	[-INF, -INF]
[±0, $\underline{fp}$]	⟵	[0,1E-5000]
[$\overline{fp}$, +inf]	⟵	[+INF, +INF]

Note: $\underline{fp}$ and $\overline{fp}$ are respectively: the smallest positive and largest representable floating–point number.

5 The empty and entire intervals

5.1 The empty interval, ∅

Three ways an empty interval can be produced are: by inputing of an empty interval (see section 21); by intersecting, `.IX.` (see section 11.2), two disjoint intervals; or by evaluating the interval enclosure of a expression using an argument that is strictly outside the expression's domain of definition.

The following are defining properties of the empty interval:

- The empty interval is a degenerate interval constant.
- The empty interval is a proper *subset*, `.PSB.` (12.6), of every other interval.
- The empty interval is *set equal*, `.SEQ.` (12.7), to itself.
- The empty interval is *disjoint*, `.DJ.` (12.1), with any interval, including itself.
- The *intersection*, `.IX.` (11.2), of the empty interval with any interval, including itself, is empty.

- Any arithmetic operation on an empty operand produces the empty interval.
- Any interval enclosure of a point arithmetic expression of one or more arguments is empty when any of its interval arguments is empty. Without loss of containment, interval enclosures of invariant (or constant) expressions can safely return the value of the invariant expression, even when the argument of the interval enclosure is empty, see [20]. Examples of invariant expressions include $f(x) = c$ and $f(x) = x^n$, for $n = 0$.
- Any point-valued (non–interval) intrinsic of one or more empty interval argument is undefined. A processor–dependent result is returned in this case. The point-valued intrinsics are: `INF`, `SUP`, `WID`, `MID`, `MIG`, `MAG`, `NDIGITS` and `INT` see sections 10, 16 and 17.

5.2 The entire interval, $\mathbb{R}^*$

For convenience of exposition the term *entire interval* is used to denote the interval $\mathbb{R}^*$. The following are defining properties of the entire interval:

- The entire interval is a non-degenerate interval constant. Two entire intervals must be treated as completely independent, see [23]. Consequently, $\mathbb{R}^* - \mathbb{R}^* = \mathbb{R}^*$ and $\mathbb{R}^*/\mathbb{R}^* = \mathbb{R}^*$ (see section 7).
- The entire interval is a proper *superset*, `.PSP.` (12.5), of every other interval.
- Two entire intervals are *set–equal*, `.SEQ.` (12.7).
- The entire interval is *disjoint*, `.DJ.` (12.1), with no other interval except the empty interval
- The *intersection*, `.IX.` (11.2), of the entire interval with any interval, X, is X.
- The result of any interval arithmetic operation on one or more entire interval operands is the range of the operation with respect to the entire interval argument(s). This is a consequence of the definition of the minimum width interval enclosure of a point operation; $X.op.Y = hull(\{\xi \mid \xi = x.op.y,\ x \in X,\ y \in Y\})$, see [20]. Example: [1,1]/[0,0] = $\{-\infty\} \cup \{+\infty\} \subset \mathbb{R}^*$
- The result of evaluating an interval enclosure of a point expression with one or more entire interval arguments is the range of the expression with respect to the entire interval argument(s). That is, $f(\mathbb{R}^*) = hull(\{y \mid y = f(x),\ x \in I\mathbb{R}^*\})$, see [20]. The extension to expressions of n-variables must include consideration of expressions resulting in indeterminate forms, see [20].

  Examples: $\mathbb{R}^{*2} = [0, +\infty] \subset \mathbb{R}^*$, $\sqrt{\mathbb{R}^*} = [0, +\infty] \subset \mathbb{R}^*$, and $\sin(\mathbb{R}^*) = [-1, 1]$.

- Any point-valued (non-interval) intrinsic of one or more entire interval arguments may or may not be defined. If not defined, a processor must return a processor-dependent result. The point-valued intrinsics are: `INF`, `SUP`, `WID`, `MID`, `MIG`, `MAG`, `NDIGITS` and `INT`, see sections 10, 16 and 17.

6 Language Extensions

The language extensions described in this section are introduced to enable interval variables to be declared, manipulated, input, and output.

6.1 Character Set

Processors are assumed to support square brackets, "`[`" and "`]`", to delimit literal interval constants, see section 6.4.

6.2 Interval related command line options

Interval features in the Sun f95 compiler are activated by means of the following command line options

- `-xinterval=(no|widestneed|strict)` is a flag to enable processing of intervals and to control permitted expression evaluation syntax.

  `"no"`, the first default, is a no-op.

  `"widestneed"`, the second default will promote all non-interval variables and literals in any mixed-mode interval expression to the widest interval data type anyplace in the expression. The details are spelled out in the interval specification document and a paper by Robert Corbett.

  `"strict"` will not permit any mixed-type or mixed-length interval expressions. All interval type and length conversions will need to be explicit, or it will be a compile-time error.

- `-xia=(widestneed|strict)` is a macro that enables the processing of interval data types and sets a suitable floating-point environment.

  If `-xia` is not mentioned, there is no expansion.

  `-xia` expands into :
  `-xinterval=widestneed`
  `-ftrap=%none`

`-fns=no`
`-fsimple=0.`

`-xia=(widestneed|strict)` expands into :
`-xinterval=(widestneed|strict)`
`-ftrap=%none`
`-fns=no`
`-fsimple=0.`

Previously set values of `-ftrap, -fns, -fsimple` are superseeded.

It is a fatal error if at the end of command line processing `-xinterval=(widestneed|strict)` is set and either -fsimple or -fns is set to any value other than

`-fsimple=0`
`-fns=no.`

If at the the end of the command line processing `-ansi` is set and `-xinterval` is set to either `widestneed` or `strict` a warning "Interval data types is a non-standard feature" is issued.

Note: `-fround = <r>`: (Set the IEEE rounding mode in effect at startup) does not interact with -xia because interval operations save and restore the rounding mode upon entry/exit.

When recognition of interval types is activated:

- Interval operators and functions become intrinsic.
- The same restrictions are imposed on the extension of interval intrinsic operators and functions as are imposed on the extension of standard intrinsic operators and functions.
- Interval specific function names (see section 6.13) are recognized.

In the remainder of this document, unless otherwise specified, the `"-xinterval=strict"` command line option is assumed to be set.

6.3 `-xtypemap=interval`

The default size of an interval variable declared only with the `INTERVAL` keyword can be changed using only the `-xtypemap` command line option[6]. The `-r8const` flag has no influence on the default size of interval types. Allowed mappings for `-xtypemap` are:

`-xtypemap=interval:128`, promoting `INTERVAL` to `INTERVAL(16)` ; and

`-xtypemap=interval:32`, demoting `INTERVAL` to `INTERVAL(4)`.

[6]Implementation of this option is not planned for current release.

6.4 Interval type

The type specifier for intervals is the keyword `INTERVAL`. An approximation method (charcterized by the kind type parameter) defines sets of values for a real data type. For interval endpoints the Fortran processor must support the same approximation methods as are used for real types. Both endpoints of an interval value must be represented using the same approximation method.

In addition to the keyword `INTERVAL`, users may specify a kind type parameter[7]. If the keyword `INTERVAL` is specified and the kind type parameter is not specified, the default kind value is the same as that for the double precision real, the type of both endpoints is double precision real, and the data entity is of type default interval.

Sun f95 also supports `INTERVAL*8`, `INTERVAL*16`, and `INTERVAL*32` type specifiers corresponding to interval values with `REAL(4)`, `REAL(8)` and `REAL(16)` endpoints. In all cases the types of both interval endpoints are the same.

The set of values that are representable using an interval constant or variable is a subset of the mathematical real interval numbers.

Intervals are opaque. That is, there is no language support provided for direct access to an interval's underlying machine representation. An intervals's infimum and supremum are accessible using the intrinsics `INF(X)` and `SUP(X)`.

If the SEQUENCE statement is present in a derived–type definition and a component has an interval type then the type is *not* a **numeric sequence type** [8].

Note: Although `COMMON, and EQUIVALENCE` variable association, and `LOC()` enable programs to access the underlying internal representation of interval components, the results of such access are processor dependent and should not be used. See also section 6.8.

Where literal constants are admitted in a program, an interval value is represented as an interval literal constant.

interval–literal–constant	is	[*endpoint*]
	or	[*left–endpoint, right–endpoint*]
left–endpoint	is	*endpoint*
right–endpoint	is	*endpoint*
endpoint	is	*signed–int–literal-constant*
	or	*signed–real–literal-constant*

An interval literal constant is specified either by one or two decimal numbers. Thus, for example, the constants `[0.1]` and `[0.1,0.1]` have the same interpretation. In either case,

[7] For the `INTERVAL` type Sun f95 supports kind type parameters equal to 4, 8 and 16

[8] No new constraint is introduced. The Fortran standard already makes these specifications for any type other than default integer, default real, default complex, or default logical.

a decimal constant's internal representation must contain the decimal constant, regardless of the number of digits in the constant.

Interval constant properties:

- An interval literal constant with a left endpoint greater than its right endpoint is invalid.
- Left and right endpoint data types may be different.
- If either endpoint is not exactly representable on a given machine, the left endpoint is rounded down and the right endpoint is rounded up to numbers known to contain the exact decimal value.
- If both endpoints are of type real but have different kind type parameters, they are both internally represented using the method of the endpoint with more decimal precision.
- If an endpoint is of type default integer, default real or double precision real, it is internally represented as a value of the type double precision real.

  An interval constant having both endpoints of type default integer, default real or double precision real, has the type default interval.
- If an endpoint's type is `INTEGER(8)`, it is internally represented using type `REAL(16)` value. If an endpoint's type `INTEGER(4)`, it is internally represented using type `REAL(8)` value. If an endpoint's type `INTEGER(1)` or `INTEGER(2)`, it is internally represented using type `REAL(4)` value.
- The kind type parameter of an interval constant is the kind type parameter value of the part with the approximation method which is applied to both parts.

Examples:

```
kind([9_8,9.0])       == 16
kind([9_8,9_8])       == 16
kind([9_4,9_4])       == 8
kind([9_2,9_2])       == 4
kind([9,9.0_16])      == 16
kind([9,9.0])         == 8
kind([9,9])           == 8
kind([9.0_4,9.0_4])   == 4
kind([1.0Q0,1.0_16]) == 16
kind([1.0_8,1.0_4])   == 8
kind([1.0e0,1.0q0])   == 16
kind([1.0e0,1])       == 8
kind([1.0q0,1])       == 16
```

6.5 Derived-type Definition

`INTERVAL` cannot be used as a derived type name. For example

```
TYPE INTERVAL
  REAL :: INF, SUP
END TYPE INTERVAL
```

is illegal.

6.6 Interval Intrinsic Expressions

An *interval expression* is used to represent interval computations. An interval expression evaluates to a scalar interval value or to an interval array value. An *interval intrinsic expression* consists of interval operands and interval operators.

Interval operands are interval constants, interval variables, interval array constructors, high precedence interval defined subexpressions, interval functions references, and (intrinsic and defined) (sub)expressions enclosed in parentheses.

A *defined expression* is composed of operands and operators, where: operands may have derived and intrinsic types; and, operators may be defined, intrinsic or extended intrinsic operators. A defined expression may also be an interval expression if it evaluates to an interval type.

A defined expression and an intrinsic expression differ either in that a defined expression has at least one derived type operand, or in that a defined expression includes either a defined operator or an extended intrinsic operator.

Operands are (as in the case of an intrinsic expression) constants, variables, array constructors, function references, and (sub)expressions enclosed in paretheses. Defined expressions may also include structure constructors used as operands.

In the integer exponenatiation operation `X**N`, where `X` is interval and `N` is an integer type variable, an integer–literal constant is allowed as an exponent only if the constant is in the range `[-MAX_INT,MAX_INT]`. This prevents containment failures caused by the truncation of integer constants that are not internally representable.

Simple interval expressions may consist only of one operand without an operator.

More complicated interval expressions consist of one or more operands processed by *interval intrinsic operators*. They may contain interval subexpressions enclosed in parentheses.

Interval intrinsic operators are:

Operator	Operation	Use	Interpretation
**	Exponenatiation	`X**Y`	Raise X to the power Y
	Multiplication	`X*Y`	Multiply X and Y
/	Division	`X/Y`	Divide X by Y
+	Addition	`X+Y`	Add X and Y
+	Identity	`+X`	Same as X (without a sign)
-	Subtraction	`X-Y`	Subtract Y from X
-	Numeric Negation	`-X`	Negate X
.IH.	Interval hull	`X.IH.Y`	Interval hull of X and Y
.IX.	Intersection	`X.IX.Y`	Intersect X and Y

Precedence of operators:

- The Unary operators `+,-` take precedence over the `**`, `*`, `+`, `-`, `.IH.`, and `.IX.`
- The operator ** takes precedence over the `*`, `+`, `-`, `.IH.`, and `.IX.` operators.
- The operators `*,/` take precedence over the `+`, `-`, `.IH.`, and `.IX.` operators.
- The operators `+,-` take precedence over the `.IH.` and `.IX.` operators.
- The operators `.IH.` and `.IX.` take precedence over the `//` operator.

Interpretation rules:

- An interval intrinsic expression is interpreted from left to right. That is, if there is an operand between two operators of the same precedence (except exponentiation), the left operator is combined with the operand.
- A parenthesized expression is treated as a data entity.
- If an expression includes operators of different precedence, the precedence of the operators controls the order of the combination of operators and operands.
- a sequence of exponentiation is combined from the right to the left; for example `[1,2]**[3,4]**[5,6]` is interpreted as $[1,2]^{([3,4]^{[5,6]})}$

6.6.1 Type, kind type parameter and shape

An interval expression evaluates to a result that is an interval. The results type, kind type parameter, shape and value are determined from those of the operands and from the interpretation of the expression.

Unary interval operator: If the operator + or - is applied to one interval operand, the type, kind type parameter and shape of the expression (and thus the type of the result) is the same as the type of the operand.

Binary interval operator:

If both operands have the default interval type, the result also has the default interval type.

With the exception of the interval ** operator with the integer exponent, an interval operator can only be applied to two operands of the same interval type and kind type parameter and thus the type and the kind type parameter of the expression (and the type of the result) is the same as the type of the operands.

If the second operand of the interval ** operator is of an integer type, the first operand can be of any interval type and the result type and the kind type parameter is that of the first operand.

A binary interval operator may be applied to two operands of different shapes (if one operand is a scalar).

Shape: If a binary interval operator is applied to two operands with the same shape, the result of the operation also has this shape. If the shapes of the operands are different (one being a scalar) the result has the shape of the array operand.

6.7 Interval Relational Intrinsic Expression

An interval relational intrinsic expression compares the results of two interval intrinsic expressions or compares (in the case of .IN. operator) the result of an real intrinsic expression with a result of an interval intrinsic expression.

An interval relational expression may appear only as an operand in a logical expression. The interval relational expression evaluates to a default logical type scalar or array value.

The list of interval relational intrinsic operators for interval data entities is:

.SP., .PSP., .SB., .PSB., .IN., .DJ., .EQ., .NE., ==, /=, .SEQ., .SNE., .SLT., .SLE., .SGT., .SGE., .CEQ., .CNE., .CLT., .CLE., .CGT., .CGE., .PEQ., .PNE., .PLT., .PLE., .PGT., .PGE.

The precedence of interval relational operators is the same as a the precedence of real relational operators, for example the .EQ. operator.

A scalar interval relational intrinsic expression evaluates to the default logical value, *true*, if and only if the operands satisfy the relation specified by the operator; otherwise the expression evaluates to the default logical value, *false*. If the operands are conformable arrays, the result of the expression is produced element–wise.

Except for the .IN. operator, the types and kind type parameters of the operands in the interval relational expression

IntervalExpression$_1$ `IntervalRelationalOperator` *IntervalExpression*$_2$

must coincide.

If the first operand of the `.IN.` operator is of any integer or real type, the second operand can be of any interval type.

The result of the interval relational expression has the default logical kind type parameter.

6.7.1 Semantics of `.EQ.` and `.NE.` operators for interval data type

The `.EQ.` (equivivalently `==`) and `.NE.` (equivivalently `/=`) relational operators can be applied to interval type operands. These operands are semanticly equivivalent to the interval set relations `.SEQ.` (12.7) and `.SNE.` (12.12), respectively.

The `.LT.`, `.LE.`, `.GT.`, `.GE.` relational operators do not accept interval type operands.

6.8 `COMMON` and `EQUIVALENCE` statements

Constraint: If an equivalence set contains an interval, all of the objects in the equivalence set must have the same type with the same kind type parameter.

Constraint: An interval variable may only be storage associated with an interval variable of the same size.

It is a desirable feature for Global Program Checking (GPC) to check for interval storage association errors.

6.9 Interval assignment: `-xinterval=strict`

Let v be an interval variable and e be an interval expression.

An interval intrinsic assignment statement $v = e$ is an assignment statement in which shapes of v and e conform.

Execution of the interval assignment $v = e$ causes the following steps to be taken:

1. All expressions used to identify the variable on the left–hand side are evaluated.
2. The interval expression on the right-hand side is evaluated.[9];
3. The result of the right–hand side is assigned (i.e stored) to the interval variable of the left–hand side.

[9] In contrast to section 9 no widest-need evaluation is used

Note: Compiling with `-xtypemap` compiler option can have an effect on the assumed type of *e*. See section 6.2.

6.10 Predefined interval operators

For the combinations of arguments listed below, interval intrinsic operators `+`, `-`, `*`, `/`, `.IH.`, `.IX.` and `**` are predefined and cannot be extended by users.

(*any* `INTERVAL` *type*, *any* `INTERVAL` *type*)
(*any* `INTERVAL` *type*, *any* `REAL` *or* `INTEGER` *type*)
(*any* `REAL` *or* `INTEGER` *type*, *any* `INTERVAL` *type*)

The interval operator `**` with the integer exponent is predefined and cannot be extended by users for the following combination of arguments:

(*any* `INTERVAL` *type*, *any* `INTEGER` *type*)

Except for the operator `.IN.` interval relational operators described in 6.7 are predefined for the combinations of arguments listed below and cannot be extended by users

(*any* `INTERVAL` *type*, *any* `INTERVAL` *type*)
(*any* `INTERVAL` *type*, *any* `REAL` *or* `INTEGER` *type*)
(*any* `REAL` *or* `INTEGER` *type*, *any* `INTERVAL` *type*)

The interval relational operator .IN. is predefined and cannot be extended by users for the following combination of arguments:

(*any* `REAL` *or* `INTEGER` *type*, *any* `INTERVAL` *type*)

6.11 Extended Interval Intrinsic Operators

If the operator specified in the `INTERFACE` statement of an operator interface block is an intrinsic interval operator (for example `.IH.`), this defines an extension of the intrinsic interval operator. An operator function for such an extended intrinsic interval operator may only extend the operator for those data types of its operands that do not belong to the data types for which this operator is predefined.

In the following example both S1 and S2 interfaces are correct because `.IH.` is not predefined for `(LOGICAL,INTERVAL(16))` and `REAL, REAL` operands,

```
MODULE M
```

```
INTERFACE OPERATOR (.IH.)
MODULE PROCEDURE S1
MODULE PROCEDURE S2
END INTERFACE

CONTAINS
REAL FUNCTION S1(X,Y)
  LOGICAL, INTENT(IN) ::  X
  INTERVAL(16), INTENT(IN) ::  Y
  S1=1
END FUNCTION S1

INTERVAL FUNCTION S2(X,Y)
  REAL, INTENT(IN) ::  X
  REAL, INTENT(IN) ::  Y
  S2=2
END FUNCTION S2
END MODULE M
```

In the following example both S3 and S4 interfaces are incorrect because `.IH.` is predefined for `(INTERVAL,INTERVAL)` and `(INTERVAL(4),INTERVAL(8))` operands.

```
MODULE M1
INTERFACE OPERATOR (.IH.)
MODULE PROCEDURE S4
MODULE PROCEDURE S3
END INTERFACE

CONTAINS
REAL FUNCTION S4(X,Y)
  INTERVAL, INTENT(IN) ::  X
  INTERVAL, INTENT(IN) ::  Y
  S4=4
END FUNCTION S4

INTERVAL FUNCTION S3(X,Y)
  INTERVAL(4), INTENT(IN) ::  X
  INTERVAL(8), INTENT(IN) ::  Y
  S3=[3]
END FUNCTION S3
```

```
END MODULE M1
```

The number of arguments of an operator function for an extended intrinsic interval operator must agree with the number of operands needed for the intrinsic operator.

For example, the following definition is incorrect:

```
MODULE M
INTERFACE OPERATOR (.IH.)
MODULE PROCEDURE S1
END INTERFACE

CONTAINS
REAL FUNCTION S1(X)
  REAL, INTENT(IN) ::  X
  S1=1
END FUNCTION S1
END MODULE M
```

A binary intrinsic interval operator can not be extended with unary operator function having an interval argument.

For example, the following definition is incorrect:

```
MODULE M
INTERFACE OPERATOR (.IH.)
MODULE PROCEDURE S1
END INTERFACE

CONTAINS
REAL FUNCTION S1(X)
  INTERVAL, INTENT(IN) ::  X
  S1=1
END FUNCTION S1
END MODULE M
```

6.12 Overloaded Interval Intrinsics Names

In a generic interface block, if the generic name specified in the `INTERFACE` statement is the name of an interval intrinsic subprogram, the user–defined specific subprograms specified

in the generic interface block extends the predefined meaning of this intrinsic subprogram. All references to subprograms having the same generic name must be unambiguous. The intrinsic subprogram is treated as a collection of specific intrinsic subprograms, the interface definitions of which are also specified in the generic interface block.

For example, the following definition is correct:

```
MODULE M
INTERFACE  WID
MODULE PROCEDURE S1
MODULE PROCEDURE S2
END INTERFACE

CONTAINS
REAL FUNCTION S1(X)
  REAL, INTENT(IN) ::  X
  S1=1
END FUNCTION S1

INTERVAL FUNCTION S2(X,Y)
  INTERVAL, INTENT(IN) ::  X
  INTERVAL, INTENT(IN) ::  Y
  S2=2
END FUNCTION S2
END MODULE M
```

In contrast, the following definition is correct.

```
 MODULE M
 INTERFACE  ABS
 MODULE PROCEDURE S1
 END INTERFACE

 CONTAINS
 INTERVAL FUNCTION S1(X)
   INTERVAL, INTENT(IN) ::  X
   S1=1
 END FUNCTION S1
 END MODULE M
```

The following definition is correct.

```
MODULE M2

INTERFACE MIN
MODULE PROCEDURE S3
END INTERFACE

CONTAINS
INTERVAL FUNCTION S3(X,Y)
  INTERVAL(4), INTENT(IN) ::  X
  INTERVAL(8), INTENT(IN) ::  Y
  S3=[3]
END FUNCTION S3
END MODULE M2
```

6.13 Specific names for interval intrinsics

The Sun f95 specific names for interval intrinsics end with the generic name of the intrinsic and start with "V", followed by " S", "D" or "Q" for arguments of type INTERVAL(4), INTERVAL(8) and INTERVAL(16), respectively.

In the current release only the following specific intrinsics are supported for the INTERVAL(16) data type: VQABS, VQAINT, VQANINT, VQINF, VQSUP, VQMID, VQMAG, VQMIG, VQISEMPTY.

To avoid name space clashes in non–interval programs, the specific names are made available by means of the command line options

"-xinterval", "-xinterval=strict" or "-xinterval=widestneed".

Examples:

Specific Name	Argument	Result
VSABS	INTERVAL(4)	INTERVAL(4)
VDABS	INTERVAL(8)	INTERVAL(8)
VQABS	INTERVAL(16)	INTERVAL(16)

7 Interval arithmetic operations

The implementation of interval arithmetic operations, $\{+, -, \times, /\}$ is based on the definition 2.

Using down and up arrows to indicate the direction of rounding in the next and subsequent operations, the following formulas can be used to perform basic interval operations (see section 8 for the definition of interval exponentiation operator):

$$
\begin{aligned}
X + Y &= \left[\downarrow \underline{x} + \underline{y}, \uparrow \overline{x} + \overline{y}\right] \\
X - Y &= \left[\downarrow \underline{x} - \overline{y}, \uparrow \overline{x} - \underline{y}\right] \\
X \times Y &= \left[\min\left(\downarrow \underline{x} \times \underline{y},\ \underline{x} \times \overline{y},\ \overline{x} \times \underline{y},\ \overline{x} \times \overline{y}\right), \max\left(\uparrow \underline{x} \times \underline{y},\ \underline{x} \times \overline{y},\ \overline{x} \times \underline{y},\ \overline{x} \times \overline{y}\right)\right] \\
X/Y &= \left[\min\left(\downarrow \underline{x}/\underline{y},\ \underline{x}/\overline{y},\ \overline{x}/\underline{y},\ \overline{x}/\overline{y}\right), \max\left(\uparrow \underline{x}/\underline{y},\ \underline{x}/\overline{y},\ \overline{x}/\underline{y},\ \overline{x}/\overline{y}\right)\right], \text{ if } 0 \notin Y \\
X/Y &= I\!R^*, \text{ if } 0 \in Y
\end{aligned}
$$

Algorithms implementing interval arithmetic operations on $I\!R^*$ for IEEE processors are contained in the Appendix A. See section 18 for interval enclosures of mathematical intrinsic functions.

> **Note:** Interval operations on valid interval operands may raise floating–point IEEE 754 exceptions. These exceptions may be ignored. If it is required to track floating–point IEEE exceptions in non–interval code, these exceptions will need to be cleared after executing interval code. See [15] for instructions to clear IEEE 754 exceptions.

8 Interval power exponentiation operators `X**n` and `X**Y`

The interval enclosure of the exponentiation operator, is more complicated than the four arithmetic operators. There are two basic cases to consider: integer exponents, x^n, and continuous exponents, x^y. The set of values an interval enclosure of X^n must contain is:

$$\{y \mid y = x^n,\ x \in X\} \ .$$

Monotonicity can be used to construct a sharp interval enclosure of the integer power intrinsic. With $n = 0$, $x^n = 1 \quad \forall x$.

The set of values an interval enclosure of X^Y must contain is

$$\{\{[\exp(y \ln(x))]\} \mid x \in X,\ y \in Y\},$$

where $\{[\exp(y \ln(x))]\}$ is the containment set of the expression $\exp(y \ln(x))$.

The result is empty if either argument is empty.

When both the base and the exponent contain zero, the result is defined to be the interval $[0, +\infty]$. This result follows from considering all the possible values satisfying the relation $\{\xi = \exp(y \ln(x)),\ x \in X,\ y \in Y,\ x \geq 0\}$, see [20].

The following table contains the containment sets for all the singularities and indeterminate forms of the intrinsic $f(x, y) = \{\exp(y \ln x)\}$:

x	y	$\{[\exp(y \ln x)]\}$
0	< 0	$+\infty$,
1	$\pm\infty$	$[0, +\infty]$
$+\infty$	0	$[0, +\infty]$
0	0	$[0, +\infty]$

9 Widest–need interval expression evaluation

Widest–need interval expression evaluation is activated by the `-xinterval=widestneed` command line option (see section 6.2).

The Fortran standard contains no accuracy requirement on the results obtained from the evaluation of floating–point or integer expression, or the evaluation of intrinsics. The Fortran standard also mandates that literal constants be treated as floating–point variables and prohibits "widest need" expression evaluation, see [2]. Without loss of containment, this combination of requirements precludes evaluating both interval expressions containing non–interval literal constants and mixed mode interval and non–interval subexpressions.

The terminology adapted from [2] provides a more formal statement of the rules for widest–need interval expression evaluation.

An arithmetic expression is *complete* if it is not an immediate operand of a generic operator. For example argument expressions of user–defined operators are complete.

The *final type* of a complete expression is the type required by the context in which it appears.

Note: Neither the interval constructor , nor interval intrinsics `INF`, `SUP`, `WID`, `MID`, `MIG`, `MAG` are generic operators.

The final type of a complete expression is the same as its result type unless the expression is: the right–hand side of an assignment; an initial value expression in a `DATA` statement; a constant expression in an `INTERVAL, PARAMETER` statement, or an operand of a relational operator.

Basic operands are constants, variables, array constructors, and functions references.

An intrinsic reference is a basic operand if the intrinsic being invoked is not one of the generic operators.

The *direct operands* of a complete expression are those basic operands that are not also contained in complete subexpressions. If the expression contains any complete subexpression, the type and kind type parameter assigned to the components of each of these subexpressions is determined separately.

9.1 Interval assignment statement

Let v be an interval variable and e be a non–complex numeric or interval expression.

An interval intrinsic assignment statement $v = e$ is an assignment statement in which shapes of v and e conform.

Execution of an interval assignment $v = e$ causes the following steps to be taken:

1. All expressions used to identify the variable on the left–hand side are evaluated.
2. The expression on the right-hand side is evaluated using widest–need interval evaluation.
3. The result of the right-hand side is converted to the kind type parameter of the left–hand side if the kind type parameters of v and e are different.
4. The (possibly converted) result of the right–hand side is assigned (i.e stored) to the interval variable of the left–hand side: v `= INTERVAL(`e`, KIND(`v`))`

The algorithm for widest–need interval evaluation of the right–hand side of the interval assignment statement can be logically split into the following steps.

1. All real operands of interval intrinsic operators are implicitly converted to containing intervals having type defined by the following rule:

   INTERVAL(*Operand*, KIND=KIND(*Operand*))

   All integer operands of interval intrinsic operators are implicitly converted to containing intervals having type defined by the following rule:

   INTERVAL(*Operand*, KIND=2*KIND(*Operand*))

2. Let e' denote the resulting interval expression implicitly constructed. The widest–need kind type parameter ($wnktp$) of the interval expression e' is determined using the following Fortran intrinsics:

   $wnktp$ = SELECTED_REAL_KIND(
   MAX(PRECISION(v),PRECISION(e'))),MAX(RANGE(v),RANGE(e'))))

3. All real, integer or interval literal constant direct operands are internally converted to a containing INTERVAL(16).

4. All direct operands are converted to containing intervals with widest–need interval type defined by the following rule:

   INTERVAL(*DirectOperand*, KIND=*wnktp*)

5. The resulting interval expression is then evaluated.

9.2 Array constructor

If an array constructor is a direct operand in an interval expression, its type is interval, and its elements are subject to widest–need interval expression evaluation.

```
REAL :: R
INTERVAL, DIMENSION(3) :: Y
Y = (/r,0.1,0.2/) ! interpretation Y = (/INTERVAL(r),[0.1],[0.2]/)
```

9.3 Interval constant expressions

Where interval constant expressions are allowed, real or integer constant expressions are allowed. Widest–need interval evaluation will apply.

9.4 Interval initialization expressions

Where interval initialization expressions are allowed, real or integer initialization expressions are allowed. Widest–need interval evaluation will apply.

9.5 Interval Relational Expressions

Let either operand of an interval relational operator be an interval expression and let the other operand be a non–complex numeric expression or an interval expression. Then the algorithm for widest–need evaluation of an interval relational intrinsic expression

exp_1 IntervaRelationalOperator exp_2

can be logically split into the following steps.

1. All real operands of interval intrinsic operators in exp_1 and exp_2 [10] are implicitly converted to containing intervals having type defined by the following rule:

   INTERVAL(*Operand*, KIND=KIND(*Operand*))

   All integer operands of interval intrinsic operators in exp_1 and exp_2 [11] are implicitly converted to containing intervals having type defined by the following rule:

   INTERVAL(*Operand*, KIND=2*KIND(*Operand*))

2. The widest–need kind type parameter (*wnktp*) of the interval expressions exp_1 and exp_2 is determined using the following Fortran intrinsics: (exp_1 and exp_2 are implicitly evaluated)

   wnktp = SELECTED_REAL_KIND(
   MAX(PRECISION(exp_1),PRECISION(exp_2))), MAX(RANGE(exp_1),RANGE(exp_2))))

3. All real, integer or interval literal constants that are direct operands in exp_1 and exp_2[12] are internally converted to containing intervals having the *wnktp* kind type parameter.

4. All direct operands in exp_1 and exp_2 [13] are converted to containing intervals with widest–need interval type defined by the following rule:

   INTERVAL(*DirectOperand*, KIND=*wnktp*)

5. The resulting interval expression exp_1 and exp_2[14] are evaluated and compared.

Thus all of the following expressions are legal and evaluate to *true*.

```
LOGICAL :: L
L= [0.1D0].SEQ.[0.1Q0]
L= [0.1].SEQ.[0.1D0]
L= 0.1.SEQ.[0.1D0]
L= 0.1.SEQ.[0.1Q0]
L= [0.1].SEQ.[0.1Q0]
```

Without widest–need evaluation only

```
L= [0.1] .SEQ. [0.1]
```

is legal.

The following expression is always illegal

```
L= 0.1 .SEQ. 0.1
```

[10] If IntervaRelationalOperator is .IN. then only exp_2 is a subject to this conversion.
[11] If IntervaRelationalOperator is .IN. then only exp_2 is a subject to this conversion.
[12] If IntervaRelationalOperator is .IN. then only exp_2 is a subject to this conversion.
[13] If IntervaRelationalOperator is .IN. then only exp_2 is a subject to this conversion.
[14] If IntervaRelationalOperator is .IN. then only exp_2 is a subject to interval evaluation.

9.6 INTERVAL versus REAL PARAMETERs

Under `-xia=widestneed` using real named constants to define interval constants must be done with care.

Real named constants are evaluated in `PARAMETER` statements i.e. the behavior of `REAL PARAMETER`s is identical to that of `REAL` variables.

In the following example interval `X` gets assigned a degenerate interval with both endpoints equal to the result of the real expression $0.1_4 + 0.2_4$.

```
REAL, PARAMETER :: PR = 0.1+0.2
REAL :: R = 0.1+0.2
INTERVAL :: X,Y
X = PR
Y = R
IF ( X .SEQ. Y ) ) PRINT *, 'Check1'
IF ( X .SEQ. INTERVAL(0.1_4+0.2_4) ) PRINT *, 'Check2'
IF ( WID(X) == 0. ) PRINT *, 'Check3'
```

An expression defining an `INTERVAL PARAMETER` is computed with the `INTERVAL(16)` precision and its result value always contain the mathematical value of the expression.

In the following example interval X gets assigned a non degenerate interval containing the mathematical value of the expression $0.1 + 0.2$.

```
INTERVAL, PARAMETER :: PY = 0.1+0.2
INTERVAL :: Y = 0.1+0.2
INTERVAL :: X
X = Y
IF ( X .SEQ. Y )  PRINT *, 'Check1'
IF ( X .SEQ. INTERVAL([0.1_16]+[0.2_16], KIND=8) ) PRINT *, 'Check2'
IF ( WID(X) /= 0. ) PRINT *, 'Check3'
```

9.7 Examples: Extended operators with widest–need evaluation

The following code illustrates how widest–need expression evaluation occurs when a predefined versus an extended version of an interval intrinsic operator is called.

```
MODULE M
INTERFACE OPERATOR (.IH.)
MODULE PROCEDURE S4
```

```
END INTERFACE

CONTAINS

INTERVAL FUNCTION S4(X,Y)
  COMPLEX, INTENT(IN) ::  X
  COMPLEX, INTENT(IN) ::  Y
  S4=[0]
END FUNCTION S4
END MODULE M

USE M
INTERVAL:: X
REAL :: R
COMPLEX :: C


X= (R-0.1).IH.(R-0.2)  ! INTRINSIC INTERVAL .IH. IS INVOKED,
                       ! WIDEST-NEED ON BOTH ARGUMENTS
X= X.IH. (R+R)         ! INTRINSIC INTERVAL .IH. IS INVOKED,
                       ! WIDEST-NEED ON BOTH ARGUMENTS
X= X.IH. (R+R+X)       ! INTRINSIC INTERVAL .IH. IS INVOKED,
                       ! WIDEST-NEED ON THE SECOND ARGUMENT
X= (R+R).IH. (R+R+X)   ! INTRINSIC INTERVAL .IH. IS INVOKED,
                       ! WIDEST-NEED ON BOTH ARGUMENTS
X = C .IH. (C + R)     ! S4 IS INVOKED, NO WIDEST-NEED
END
```

The following code illustrates how widest–need expression evaluation occurs when a user-defined operator is called.

The following is the expected behavior for the current release:

```
MODULE M

INTERFACE OPERATOR (.AA.)
MODULE PROCEDURE S1
MODULE PROCEDURE S2
END INTERFACE

CONTAINS
```

```
INTERVAL FUNCTION S1(X,Y)
  INTERVAL, INTENT(IN) ::  X
  REAL, INTENT(IN) ::  Y
  S1=[0]
END FUNCTION S1

INTERVAL FUNCTION S2(X,Y)
  INTERVAL, INTENT(IN) ::  X
  INTERVAL, INTENT(IN) ::  Y
  S2=[0]
END FUNCTION S2
END MODULE M

USE M
INTERVAL:: X
REAL :: R

X= X.AA.(R+R) ! S1 IS INVOKED [15]

X= X.AA.X ! S2 IS INVOKED
X= R.AA.R ! Error :  No operator .AA. with (REAL,REAL) arguments is defined.
X= R.AA.X ! Error:  No operator .AA. with (REAL,INTERVAL) arguments is defined.
END
```

9.8 Examples: Widest–need interval expression evaluation

To guarantee containment of pair of REAL variables, the interval hull, `.IH.`, operator is used:

```
REAL(16) EPS, A
INTERVAL X
X= (A-EPS) .IH. (A+EPS)
```

In the following examples, assume these variables have been declared:

```
INTEGER(2) ::  I,   INTEGER(4) ::  DI,   INTEGER(8) ::  QI
REAL(4)::  R,       REAL(8) ::  DR,      REAL(16) ::  QR
INTERVAL(4) ::  X,  INTERVAL(8) ::  DX,  INTERVAL(16) ::  QX
```

[15] A warning indicating a potential violation of containment is issued

```
Expression:      X = DX + R * QI
Interpretation:  X = SINTERVAL(QINTERVAL(DX)+QINTERVAL(R)*QINTERVAL(QI))

Expression:      DX = R + WID(MAX(DI, QR+ SINTERVAL(R, I)))
Interpretation:  DX = DINTERVAL(QINTERVAL(R)+
                 QINTERVAL(WID(MAX(QINTERVAL(DI),
                 QINTERVAL(QR)+ QINTERVAL(SINTERVAL(R, I)))))

Expression:      DX = QR + WID(MAX(DI, DR+ SINTERVAL(R, I)))
Interpretation:  DX = DINTERVAL(QINTERVAL(QR)+
                 QINTERVAL(WID(MAX(DINTERVAL(DI), DINTERVAL(DR)+
                 DINTERVAL(SINTERVAL(R, I))))))

Expression:      DX = R + DINTERVAL(DR, R + WID(X + QX))
Interpretation:  DX = DINTERVAL(R)+DINTERVAL(QREAL(DR), QREAL(R)+
                 WID(QINTERVAL(X)+QX))

Expression:      DX = USER_FUNCTION_RETURNING_INTERVAL(R + WID(QX))
Interpretation:  DX = DINTERVAL(USER_FUNCTION_RETURNING_INTERVAL(QREAL(R) +
                 WID(QX)))

Expression:      DX = QX + DCOS( R * R + MID(X+X * DSIN(DR)))
Interpretation:  DX = DINTERVAL(QX+QINTERVAL(DCOS(DREAL(R)*DREAL(R)+
                 MID(DINTERVAL(X)+ DINTERVAL(X)*DINTERVAL(DSIN(DR))))))
Expression:      DX = DR .IH. R
Interpretation:  DX = DINTERVAL(DR) .IH. DINTERVAL(R)
```

If default interval type is `INTERVAL(16)` then:

```
Expression:      DX = R + INTERVAL(DR, R + WID(X + QX))
Interpretation:  DX = QINTERVAL(R)+QINTERVAL(QREAL(DR), QREAL(R)+
                 WID(QINTERVAL(X)+ QX))
```

Note: Using `DINTERVAL` constructor in the statement

```
DX = R + DINTERVAL(DR, R + WID(X + DX))
```

insulates its arguments from widest-need evaluation. To guarantee containment the interval hull, `.IH.`, and widest-need expression evaluation are required

```
DX = R +DR .IH. ( R + WID(X + DX))
```

Descriptions of intrinsic algorithms

The description of each Fortran intrinsic and operation contains an algorithm defining the logic of a possible impelementation. These algorithms are not a part of the specification body. They are provided only for IEEE 754 compliant processors.

The algorithms that rely on the `MIN`" and `MAX` intrinsics, (for example the `.IX.`, `MIG`, `MAG`, `ABS`, `MAX`, and `MIN` intrinsics), are guaranteed to correctly return an empty result for an empty argument, only if the `MIN` and `MAX` propagate the $NaN_\emptyset$ pattern:

$\min\{x, NaN_\emptyset\} = NaN_\emptyset$, for all floating–point x
$\max\{x, NaN_\emptyset\} = NaN_\emptyset$, for all floating–point x

10 Interval set intrinsics.

In the description of algorithms the notation $X = [\underline{x}, \overline{x}]$ for the components of the interval X is taken for granted.

10.1 `INF(X)`

Description. Infimum [16] of an interval.

Class. Elemental function.

Argument X is of type interval.

Result characteristics. The result is of type real. The kind type parameter is that of X.

Result value. The result is $\underline{x}$. Because `INF`($\emptyset$) is undefined, if X is empty the result is a quiet NaN.

Algorithm. If .NOT. ISEMPTY(X) then $\underline{x}$ else NaN ;

10.2 `ISEMPTY(X)`

Description. Tests if X is the empty interval.

$$ISEMPTY(X) \equiv (X = \emptyset)$$

Class. Inquiry function.

[16] The largest number that is less than or equal to each of a given set of real numbers.

Argument X is of type interval.

Result characteristics. Default logical scalar.

Result value. If X is empty, the result is $true$ otherwise the result is $false$.

Algorithm. ISNAN($\underline{x}$) and ISNAN($\overline{x}$)

10.3 SUP(X)

Description. Supremum [17] of an interval.

Class. Elemental function.

Argument. X is of type interval.

Result characteristics. The result is of type real. The kind type parameter is that of X.

Result value. The result is $\overline{x}$.

Because SUP($\emptyset$) is undefined, if X is empty the result is a quiet NaN.

Algorithm. If .NOT. ISEMPTY(X) then $\overline{x}$ else NaN ;

11 Set operations

11.1 Interval Hull (X .IH. Y)

Description. Interval hull of two intervals

$$X.\mathrm{IH}.Y \equiv \begin{cases} \text{if } X = \emptyset & : \quad Y \\ \text{else if } Y = \emptyset & : \quad X \\ \text{else} & : \quad [\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}] \end{cases}$$

Argument. X and Y are of type interval and have the same kind type parameter.

Result characteristics. Same as X.

Result value. If one of the operands of the interval hull operator is the empty interval then the result is the other operand.

The interval result is an enclosure of the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

Algorithm.

[17] The smallest number that is greater than or equal to each of a given set of real numbers.

$$\begin{array}{lcl} \text{if } ISEMPTY(X) & : & [\underline{y}, \overline{y}] \\ \text{else if } ISEMPTY(Y) & : & [\underline{x}, \overline{x}] \\ \text{else} & : & [\min\{\underline{x}, \underline{y}\}, \max\{\overline{x}, \overline{y}\}] \end{array}$$

11.2 Intersection (X .IX. Y)

Description. Intersection of two intervals.

$$X \cap\ Y \equiv \{x : x \in X \text{ and } x \in Y\}$$

Arguments. X and Y are of type interval and have the same kind type parameter.

Result characteristics. Same as X.

Result value. If either operand of the interval intersection operator is the empty interval then the result is the empty interval.

The interval result is an enclosure of the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

Algorithm.

$$\begin{array}{lcl} [\max\{\underline{x}, \underline{y}\}, \min\{\overline{x}, \overline{y}\}] & : & \text{if } \max\{\underline{x}, \underline{y}\} \leq \min\{\overline{x}, \overline{y}\} \\ \emptyset & : & \text{otherwise} \end{array}$$

12 Set relations

For a relation $.op. \in \{<, >, \leq, \geq, =\}$ between two points x and y, the corresponding *set* relation $.Sop.$ between two intervals X and Y is

$$X \text{ .Sop.} Y \ \equiv\ (\forall x \in X,\ \exists y \in Y :\ x.op.y) \text{ and } (\forall y \in Y, \exists x \in X :\ x.op.y)$$

For the relation $\neq$ between two points x and y, the corresponding *set* relation $.SNE.$ between two intervals X and Y is

$$X \ \text{ .SNE. } Y \ \equiv\ (\exists x \in X,\ \forall y \in Y :\ x \neq y) \text{ or } (\exists y \in Y, \forall x \in X :\ x \neq y)$$

These definitions apply when the operands are independent or dependent: $(X.Sop.X)$.

The following table summarizes the results for X.Sop.X expressions, where .Sop. is one of the set relational oparators: `.SLT.`, `.SGT.`, `.SLE.`, `.SGE.`, `.SEQ.` or `.SNE.`

	$X \neq \emptyset$		$X = \emptyset$
X.Sop.X	$\underline{x} = \overline{x}$	$\underline{x} \neq \overline{x}$	
`X.SLT.X`	*false*	*false*	*false*
`X.SGT.X`	*false*	*false*	*false*
`X.SLE.X`	*true*	*true*	*true*
`X.SGE.X`	*true*	*true*	*true*
`X.SEQ.X`	*true*	*true*	*true*
`X.SNE.X`	*false*	*false*	*false*

Compile–time optimization can be applied to all of these expressions, because their results are invariant with respect to the value of X.

Note: The above optimizations can be applied to variables, constants or expressions.

12.1 Disjoint `(X .DJ. Y)`

Description. Tests if two intervals are disjoint.

$$X \text{ .DJ. } Y \equiv (\forall x \in X,\ \forall y \in Y :\ x \neq y)$$

The result is true if one or both arguments is empty.

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} > \overline{y}$ or $\overline{x} < \underline{y}$) or either X or Y or both is empty. Otherwise the result is *false*.

Algorithm. not ($\underline{x} \leq \overline{y}$ and $\underline{y} \leq \overline{x}$) [18]

Possible compile-time action.

> From the definition of the disjoint operator it follows that `X .DJ. X` is *false* if `X` is not empty. The proposed algorithm correctly enforces this result at run time, but the check "`ISEMPTY(X)`" may be substituted for `X .DJ. X` at compile time.

[18] To get the desired *true* result if one or both arguments is empty the complement of ($\underline{x} \leq \overline{y}$ and $\underline{y} \leq \overline{x}$) rather than ($\underline{x} > \overline{y}$ or $\overline{x} < \underline{y}$) must be used. This is because the equality test invloving a NaN operand is always *false* and the inequality test is always *true*. That is NaN== NaN is *false* and NaN /= NaN is *true*.

12.2 In (R .IN. Y)

Description. Tests if the number is contained in the interval.

$$R \in Y \equiv (\exists y \in Y : \ y = R)$$

Arguments.
R is of type integer or real.
Y is of type interval.

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{y} \leq R$ and $R \leq \overline{y}$) otherwise the result is $false$. The result is $false$ if Y is empty .

Algorithm. $\underline{y} \leq R$ and $R \leq \overline{y}$

12.3 Interior (X .INT. Y).

Description. Tests if X is in interior of Y

$$X.\text{INT}.Y \equiv (X = \emptyset) \text{ or } (\forall x \in X, \ \exists y' \in Y, \exists y'' \in Y : \ y' < x < y'')$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{y} < \underline{x}$ and $\overline{x} < \overline{y}$) or X is empty[19]. Otherwise the result is $false$.

Algorithm. ($\underline{y} < \underline{x}$ and $\overline{x} < \overline{y}$) or ISEMPTY(X)

12.4 Proper subset (X .PSB. Y)

Description. Tests if X is a proper subset of Y

$$X \subset Y \equiv X \subseteq Y \text{ and } X \text{ .SNE. } Y$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

[19]It is worth mentioning that in contrast to $\emptyset$.PSP.$\emptyset = false$ (see section 12.5), $\emptyset$.INT.$\emptyset = true$. The interior of a set in topological space is a union of all open subsets of the set. An empty set is open and therefore is a subset of the interior of the empty set.

Result value. The result is *true* if X is a subset of Y and X is set–not–equal (12.12) to Y. Otherwise the result has the value $false$.

Algorithm.
($\underline{x} \geq \underline{y}$ and $\overline{x} \leq \overline{y}$ and ($\underline{x} > \underline{y}$ or $\overline{x} < \overline{y}$)) or(ISEMPTY(X) and not ISEMPTY (Y))

12.5 Proper superset `(X .PSP. Y)`.

Description. Tests if X is a proper superset of Y

$$X \supset Y \equiv X \supseteq Y \text{ and } X \text{ .SNE. } Y$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if X is a superset of Y and Y is set–not–equal (12.12) to X. Otherwise the result has the value $false$.

Algorithm.
($\underline{x} \leq \underline{y}$ and $\overline{y} \leq \overline{x}$ and ($\underline{x} < \underline{y}$ or $\overline{y} < \overline{x}$)) or (ISEMPTY(Y) and not ISEMPTY (X))

12.6 Subset `(X .SB. Y)`.

Description. Tests if X is a subset of Y

$$X \subseteq Y \equiv (X = \emptyset) \text{ or } (\forall x \in X,\ \exists y' \in Y, \exists y'' \in Y :\ y' \leq x \leq y'')$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{y} \leq \underline{x}$ and $\overline{x} \leq \overline{y}$) or X is empty). Otherwise the result is $false$.

Algorithm. ($\underline{y} \leq \underline{x}$ and $\overline{x} \leq \overline{y}$) or ISEMPTY(X)

12.7 Set–equal `(X .SEQ. Y)`

Description. Tests if two intervals are set–equal.

$$X \text{ .SEQ.} Y \equiv (\forall x \in X,\ \exists y \in Y :\ x = y) \text{ and } (\forall y \in Y, \exists x \in X :\ x = y)$$

Any interval including the empty interval is set–equal to itself.

$$X.\text{SEQ}.X \equiv \ TRUE$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $\underline{x} = \underline{y}$ and $\overline{x} = \overline{y}$ or both arguments are empty. Otherwise the result is $false$.

Algorithm. $\underline{x} = \underline{y}$ and $\overline{x} = \overline{y}$ or $ISEMPTY(X)$ and $ISEMPTY(Y)$

12.8 Set–greater-or-equal (X .SGE. Y)

Description. Tests if one interval is set–greater–or–equal to another.

$$X \ .\text{SGE}.Y \ \equiv \ (\forall x \in X, \ \exists y \in Y : \ x \geq y) \text{ and } (\forall y \in Y, \exists x \in X : \ x \geq y)$$

$$X \ .\text{SGE}. \ X \equiv \ TRUE$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} \geq \underline{y}$ and $\overline{x} \geq \overline{y}$) otherwise the result is $false$. The result is $false$ if only one of the arguments is empty. The result is *true* if both arguments are empty.

Algorithm.

$\underline{x} \geq \underline{y}$ and $\overline{x} \geq \overline{y}$ or $ISEMPTY(X)$ and $ISEMPTY(Y)$

12.9 Set–greater (X .SGT. Y)

Description. Tests if one interval is set–greater than another.

$$X \ .\text{SGT}.Y \ \equiv \ (\forall x \in X, \ \exists y \in Y : x > y) \text{ and } (\forall y \in Y, \exists x \in X : x > y)$$

$$X.\text{SGT}. \ X \equiv \ FALSE$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} > \underline{y}$ and $\overline{x} > \overline{y}$) otherwise the result is $false$. The result is $false$ if one or both of the arguments is empty

Algorithm. $\underline{x} > \underline{y}$ and $\overline{x} > \overline{y}$

12.10 Set–less-or-equal (X .SLE. Y)

Description. Tests if one interval is set–less–or–equal to another.

$$X \text{ .SLE.} Y \equiv (\forall x \in X,\ \exists y \in Y :\ x \leq y) \text{ and } (\forall y \in Y,\ \exists x \in X :\ x \leq y)$$

$$X\text{.SLE.}X \equiv TRUE$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} \leq \underline{y}$ and $\overline{x} \leq \overline{y}$) otherwise the result is $false$. The result is $false$ if only one of the arguments is empty. The result is *true* if both arguments are empty.

Algorithm. $\underline{x} \leq \underline{y}$ and $\overline{x} \leq \overline{y}$ or $ISEMPTY(X)$ and $ISEMPTY(Y)$

12.11 Set–less (X .SLT. Y)

Description. Tests if one interval is set–less than another.

$$X \text{ .SLT.} Y \equiv (\forall x \in X,\ \exists y \in Y :\ x < y) \text{ and } (\forall y \in Y, \exists x \in X :\ x < y)$$

$$X\text{.SLT.}X \equiv FALSE$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} < \underline{y}$ and $\overline{x} < \overline{y}$) and otherwise the result is $false$. The result is $false$ if one or both of the arguments is empty

Algorithm. $\underline{x} < \underline{y}$ and $\overline{x} < \overline{y}$

12.12 Set–not–equal (X .SNE. Y)

Description. Tests if two intervals are not set–equal.

$$X \text{ .SNE. } Y \equiv (\exists x \in X,\ \forall y \in Y :\ x \neq y) \text{ or } (\exists y \in Y, \forall x \in X :\ x \neq y)$$

Any interval including the empty interval is set–equal to itself, therefore

$$X \text{ .SNE. } X \equiv FALSE.$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} \neq \underline{y}$ or $\overline{x} \neq \overline{y}$) or one or the other argument is the empty interval. Otherwise the result is false.

Algorithm. not ($\underline{x} = \underline{y}$ and $\overline{x} = \overline{y}$ or $ISEMPTY(X)$ and $ISEMPTY(Y)$) [20]

12.13 Superset (X .SP. Y).

Description. Tests if X is a superset of Y.

$$X \supseteq Y \equiv (Y = \emptyset) \text{ or } (\forall y \in Y,\ \exists x' \in X,\ \exists x'' \in X :\ x' \leq y \leq x'')$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} \leq \underline{y}$ and $\overline{y} \leq \overline{x}$) or Y is the empty interval. Otherwise the result is $false$.

Algorithm. ($\underline{x} \leq \underline{y}$ and $\overline{y} \leq \overline{x}$) or ISEMPTY(Y)

13 Certainly relations

For a relation $.op. \in \{<, >, \leq, \geq, =, \neq\}$ between two points x and y, the corresponding *certainly true* relation $.Cop.$ between two intervals X and Y is

$$X.Cop.Y = TRUE \text{ iff } (\forall x \in X,\ \forall y \in Y :\ x.op.y = TRUE)$$

If the empty interval is an operand of a *certainly* relation then the result is $false$. The one exception is the *certainly–not–equal* relation (13.6), which is *true* in this case.

When both operands are completely dependent, $X.Cop.X$ is defined :

$$X.Cop.X = TRUE \text{ iff } (\forall x' \in X,\ \forall x'' \in X :\ x'.op.x'' = TRUE).$$

The following table summarizes the results for X.Cop.X expressions.

[20] In contrast to the algorithm in [7] the .SNE. operator's algorithm must use the complement of the .SEQ. relation (12.7). Otherwise if both operands are empty the required $false$ result will not be produced. This is because the equality test invloving a NaN operand is always $false$ and the inequality test is always *true*. That is, NaN== NaN is $false$ and NaN $\neq$ NaN is *true*.

	$X \neq \emptyset$		$X = \emptyset$
X.Cop.X	$\underline{x} = \overline{x}$	$\underline{x} \neq \overline{x}$	
`X.CLT.X`	$false$	$false$	$false$
`X.CGT.X`	$false$	$false$	$false$
`X.CLE.X`	$true$	$false$	$false$
`X.CGE.X`	$true$	$false$	$false$
`X.CEQ.X`	$true$	$false$	$false$
`X.CNE.X`	$false$	$false$	$true$

Compile–time optimization can be applied to `X.CLT.X` and `X.CGT.X` expressions, because their results are invariant with respect to the value of X.

13.1 Certainly–equal `(X .CEQ. Y)`

Description. Tests if one interval is certainly–equal to another [21]

$$X \;=\; Y \;\equiv\; (\forall x \in X,\ \forall y \in Y :\ x = y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\overline{x} \leq \underline{y}$ and $\underline{x} \geq \overline{y}$). Otherwise the result is $false$. If one or both arguments is empty, the result is $false$.

Algorithm. $\overline{x} \leq \underline{y}$ and $\underline{x} \geq \overline{y}$

13.2 Certainly–greater–or–equal `(X .CGE. Y)`

Description. Tests if one interval is certainly–greater–or–equal to another.

$$X \;\geq\; Y \;\equiv\; (\forall x \in X,\ \forall y \in Y :\ x \geq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\underline{x} \geq \overline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\underline{x} \geq \overline{y}$

[21] `(X .CEQ. Y)` = *true* implies that X or Y can be substitutes for one another in any expression. This is not possible if `(X .SEQ. Y)`= *true*, but `(X .CEQ. Y)` = $false$.

13.3 Certainly greater (X .CGT. Y)

Description. Tests if one interval is certainly–greater than another.

$$X > Y \equiv (\forall x \in X,\ \forall y \in Y :\ x > y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\underline{x} > \overline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\underline{x} > \overline{y}$

13.4 Certainly–less–or–equal (X .CLE. Y)

Description. Tests if one interval is certainly–less–or–equal to another.

$$X \leq Y \equiv (\forall x \in X,\ \forall y \in Y :\ x \leq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\overline{x} \leq \underline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm.$\overline{x} \leq \underline{y}$

13.5 Certainly–less (X .CLT. Y)

Description. Tests if one interval is certainly–less than another.

$$X < Y \equiv (\forall x \in X,\ \forall y \in Y :\ x < y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\overline{x} < \underline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\overline{x} < \underline{y}$

13.6 Certainly–not–equal (X .CNE. Y)

Description. Tests if one interval is certainly–not–equal to another [22]

$$X \neq Y \equiv (\forall x \in X,\ \forall y \in Y :\ x \neq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} > \overline{y}$ or $\underline{y} > \overline{x}$) . Otherwise the result is $false$. If one or both arguments is empty, the result is *true*.

Algorithm. not ($\underline{x} \leq \overline{y}$ and $\underline{y} \leq \overline{x}$) [23]

14 Possibly relations

For a relation *.op.* $\in \{<, >, \leq, \geq, =, \neq\}$ between two points x and y, the corresponding *possibly true* relation $.Pop.$ between two intervals X and Y is defined as follows:

$$X.Pop.Y = TRUE \text{ iff } (\exists x \in X,\ \exists y \in Y :\ x.op.y = TRUE)$$

If the empty interval is an operand of a *possibly* relation then the result is $false$. The one exception is the *possibly not equal* relation (14.6) which is *true* in this case.

Whenever both intreval operands are completely dependent, $X.Pop.X$ is defined :

$$X.Pop.X = TRUE \text{ iff } (\exists x' \in X,\ \exists x'' \in X :\ x'.op.x'' = TRUE).$$

The following table summarizes the results for X.Pop.X expressions.

	$X \neq \emptyset$		$X = \emptyset$
X.Pop.X	$\underline{x} = \overline{x}$	$\underline{x} \neq \overline{x}$	
X.PLT.X	$false$	$true$	$false$
X.PGT.X	$false$	$true$	$false$
X.PLE.X	$true$	$true$	$false$
X.PGE.X	$true$	$true$	$false$
X.PEQ.X	$true$	$true$	$false$
X.PNE.X	$false$	$true$	$true$

Compile–time optimization cannot be applied to any of these expressions, because none of their results are invariant with respect to the value of X.

[22] The semantic and the algorithm for the .CNE. operator are equivalent to those of the .DJ. operator.

[23] To get the desired *true* result the complement of ($\underline{x} \leq \overline{y}$ and $\underline{y} \leq \overline{x}$) rather than ($\underline{x} > \overline{y}$ or $\underline{y} > \overline{x}$) must be used. This is because the equality test invloving a NaN operand is always $false$ and the inequality test is always *true*. That is NaN== NaN is $false$ and NaN $\neq$ NaN is *true*.

14.1 Possibly–equal (X .PEQ. Y)

Description. Tests if one interval is possibly–equal to another.

$$X \text{ .PEQ.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x = y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\underline{x} \leq \overline{y}$ and $\overline{x} \geq \underline{y}$) . Otherwise the result is $false$. If one or both arguments is empty, the result is $false$.

Algorithm.$\underline{x} \leq \overline{y}$ and $\overline{x} \geq \underline{y}$

14.2 Possibly–greater–or–equal (X .PGE. Y)

Description. Tests if one interval is possibly–greater–or–equal to another.

$$X \text{ .PGE.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x \geq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\overline{x} \geq \underline{y}$) . Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm.$\overline{x} \geq \underline{y}$

14.3 Possibly–greater (X .PGT. Y)

Description. Tests if one interval is possibly–greater than another.

$$X \text{ .PGT.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x > y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if ($\overline{x} > \underline{y}$) . Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\overline{x} > \underline{y}$

14.4 Possibly–less–or–equal (X .PLE. Y)

Description. Tests if one interval is possibly–less-or-equal to another.

$$X \text{ .PLE.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x \leq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\underline{x} \leq \overline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\underline{x} \leq \overline{y}$

14.5 Possibly–less (X .PLT. Y)

Description. Tests if one interval is possibly–less than another.

$$X \text{ .PLT.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x < y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\underline{x} < \overline{y})$. Otherwise the result is $false$. If one or both of the arguments is empty, the result is $false$.

Algorithm. $\underline{x} < \overline{y}$

14.6 Possibly–not–equal (X .PNE. Y)

Description. Tests if one interval is possibly–not–equal to another.

$$X \text{ .PNE.} Y \equiv (\exists x \in X,\ \exists y \in Y :\ x \neq y)$$

Arguments. X and Y are of type interval

Result characteristics. Default logical scalar.

Result value. The result is *true* if $(\overline{x} > \underline{y}$ or $\underline{x} < \overline{y})$. Otherwise the result is $false$. If one or both arguments is empty, the result is *true*.

Algorithm. not $(\overline{x} \leq \underline{y}$ and $\underline{x} \geq \overline{y})$ [24]

[24]To get the desired *true* result if one or both arguments is emptythe complement of the algorithm for the certainly equal relation rather than $(\overline{x} > \underline{y}$ or $\underline{x} < \overline{y})$ must be used. This is because the equality test invloving a NaN operand is always $false$ and the inequality test is always *true*. That is NaN== NaN is $false$ and NaN /= NaN is *true*.

15 Precedence of Operators

The precedence of interval operations determines the order in which the operands are combined, unless the order is changed by the use of parentheses. The precedence order of interval operators is summarized in the following table.

In the absence of parentheses, if there is more than one operator in an expression, then the operators are applied in the order of precedence. If the operators are of equal precedence, they are applied left to right.

Interval relational operators have the same precedence as the other Fortran relational operators and therefore have higher precedence than logical operators.

Category of operation	Operators	Precedence
Numeric	`**`	Highest
Numeric	`*` or `/`	
Numeric	unary `+` or `-`	·
Numeric	binary `+` or `-`	·
Set	`.IX.`, `.IH.`	·
Relational	`.SP.`, `.PSP.`, `.SB.`, `.PSB.`, `.IN.`, `.DJ.`,	
	`.EQ.`, `.NE.`, `==`, `/=`,	
	`.SEQ.`, `.SNE.`, `.SLT.`, `.SLE.`, `.SGT.`, `.SGE.`,	
	`.CEQ.`, `.CNE.`, `.CLT.`, `.CLE.`, `.CGT.`, `.CGE.`,	
	`.PEQ.`, `.PNE.`, `.PLT.`, `.PLE.`, `.PGT.`, `.PGE.`	Lowest

Examples:

Expression	Interpretation
`X + Y .IX. Z`	`(X+Y) .IX. Z`
`X + Y .SB. Z`	`(X+Y) .SB. Z`
`X .SB. Y .IX. Z`	`X .SB. (Y .IX. Z )`
`X .IX. Y .SEQ. Z`	`(X .IX. Y) .SEQ. Z`

16 Special interval intrinsics

16.1 Absolute value: `ABS(X)`

Description. Range of absolute value.

$$ABS(X) \supseteq \{|x| \mid x \in X\}$$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

If the argument is empty, the result is the empty interval.

Algorithm. $[MIG(X), MAG(X)]$

16.2 Magnitude: `MAG(X)`

Description. The greatest absolute value in the non-empty interval.
$\max\{|x| \mid x \in X\}$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. The result is of type real.

Result value. $\max\{|\underline{x}|, |\overline{x}|\}$.

`MAG(`$\emptyset$`)= NaN, MAG(` $\mathbb{R}^*$`)= +INF`

Algorithm. $\max\{|\underline{x}|, |\overline{x}|\}$

16.3 Maximum: `MAX(X1, X2 [, X3, ...])`

Description. Range of maximum.

The containment set for $\max(X_1, \ldots, X_n))$ is : $\{\max(X_1, \ldots, X_n))\} =$
$\{\xi \mid \xi = \max(x_1, \ldots, x_n),\ x_i \in X_i\} = [\sup(\text{hull}(\underline{x_1}, \ldots, \underline{x_n})), \sup(\text{hull}(\overline{x_1}, \ldots, \overline{x_n}))]$

The implementation of the `MAX` intrinsic must satisfy: `MAX(X1, X2[X3,...])` $\supseteq \{\max(X_1, ..., X_n)\}$

Class. Elemental function.

Arguments. The arguments are of type interval and all have the same kind type parameter.

Result characteristics. The result is of type interval. The kind type parameter is that of the arguments.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description. [25]

[25] As long as at least one argument of the `MAX` intrinsic is not empty, the returned value will not be empty. Only if all arguments are empty is $\max(X_1, \ldots, X_n) = \emptyset$.

16.4 Midpoint: `MID(X)`

Description. Midpoint of the non-empty interval.

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. The result is of type real.

Result value. The result is a processor–dependent approximation of $(\underline{x} + \overline{x})/2$. For non-empty argument the result must fulfill the property $\underline{x} \leq MID(X) \leq \overline{x}$.

`MID(∅)= NaN, MID( IR*)= 0`

Algorithm. if $\underline{x} = \overline{x}$ then $\underline{x}$ else if $\mid \underline{x} \mid = \mid \overline{x} \mid$ then 0, else $(0.5 * \underline{x} + 0.5 * \overline{x})$

16.5 Mignitude: `MIG(X)`

Description. The smallest absolute value in the non-empty interval.
$\min\{|x| \mid x \in X\}$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. The result is of type real.

Result value. The result is $\min\{|\underline{x}|, |\overline{x}|\}$, if $0 \notin X$; otherwise the result is 0.

`MIG(∅)= NaN, MIG( IR*)= 0`

Algorithm.

$$\begin{array}{lll} \min\{|\underline{x}|, |\overline{x}|\} & : & \text{if } 0 \notin X \\ 0 & : & \text{otherwise.} \end{array}$$

16.6 Minimum: `MIN(X1, X2 [, X3, ...])`

Description. Range of minimum.

The containment set for $\min(X_1, \ldots, X_n))$ is : $\{\min(x_1, \ldots, x_n))\} =$
$\{\xi \mid \xi = \min(x_1, \ldots, x_n),\ x_i \in X_i\} = [\inf(\text{hull}(\underline{x_1}, \ldots, \underline{x_n})), \inf(\text{hull}(\overline{x_1}, \ldots, \overline{x_n}))]$

The implementation of the `MIN` intrinsic must satisfy: `MIN(X1, X2[X3,...])` $\supseteq \{\min(X_1, ..., X_n)\}$

Class. Elemental function.

Arguments. The arguments are of type interval and all have the same kind type parameter.

Result characteristics. The result is of type interval. The kind type parameter is that of the arguments.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description. [26]

16.7 NDIGITS(X)

Description. Maximum number of significant decimal digits in the single-number representation (see section 21.3) of the non-empty interval X.

Class. Elemental function.

Arguments. X is of type interval.

Result characteristics. Default integer scalar.

Result value. The integer result is the maximum number of significant digits in a decimal number y such that X is contained in an interval whose endpoints are defined by adding and subtracting 1 unit in the last digit (uld) of y.

For example $2.3 + [-1, 1]_{uld} = [2.2, 2.4]$ and $2.30 + [-1, 1]_{uld} = [2.29, 2.31]$

If X is a degenerate non-empty interval ($\underline{x} = \overline{x}$), then the result is `MAX_INT` (the largest representable integer).

`NDIGITS (`$\emptyset$`) = NDIGITS(` $\mathbb{R}^*$`) =` 0

Examples

```
NDIGITS ([0.0, .5]) = 1
NDIGITS ([1, 1]) = MAX_INT
NDIGITS ([2.345690 E+7, 2.345679E+7]) = 7
```

16.8 Width: WID(X)

Description. Width of the non-empty interval.

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. The result is of type real.

[26] As long as at least one argument of the `MIN` intrinsic is not empty, the returned value will not be empty. Only if all arguments are empty is $\min(X_1, ..., X_n) = \emptyset$.

Result value. The result is a processor–dependent upper bound on $(\overline{x} - \underline{x})$. That is, if $(\overline{x} - \underline{x})$ is not machine representable then it is upwardly rounded.

`WID(∅)= NaN, WID( IR*)= +INF`

Algorithm. $(\overline{x} - \underline{x})$ rounded up.

17 INT(X [, KIND])

Description. Convert the midpoint of the non–empty interval argument to integer type.

Class. Elemental function.

Arguments. X is of type interval

KIND (optional) is a scalar integer initialization expression.

Result characteristics. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, kind type parameter is that of default integer type.

Result value. The result value is `INT( MID (X))`.

`INT (∅) = INT(NaN), INT( IR*) = 0`

18 Interval enclosures of mathematical functions

All Fortran intrinsics that accept real data accept interval data.

The results from [20] eliminate any restriction on the domain of any interval enclosures of real expressions. Points at which real functions are not defined, need not be excluded from the domain of the function's interval enclosures. Extended intervals can always be used to return an interval enclosure.

18.1 ACOS(X)

Description. Interval enclosure of the inverse cosine intrinsic over an interval.

$$\text{acos}(X) \supseteq \ \{\{[\text{acos}(x)]\} \mid x \in X \cap \ [-1, 1]\}.$$

Special values. $\text{acos}(\mathbb{R}^*) = [0, \pi]$.

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

18.2 AINT(X [, KIND])

Description. Interval enclosure of the truncation to a whole number intrinsic over an interval.

$$AINT(X) \supseteq \{\{AINT(x)\} \mid x \in X\}.$$

Class. Elemental function.

Arguments. X is of type interval.

KIND (optional) is a scalar integer initialization expression.

Result characteristics. The result is of type interval. If KIND is present, the kind type parameter is that specified by KIND; otherwise, kind type parameter is that of X.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if argument is empty.

18.3 ANINT(X [, KIND])

Description. Interval enclosure of the nearest whole number intrinsic over an interval.

$$ANINT(X) \supseteq \{\{ANINT(x)\} \mid x \in X\}.$$

Class. Elemental function.

Arguments. X is of type interval.

KIND (optional) is a scalar integer initialization expression.

Result characteristics. The result is of type interval. If KIND is present, the kind type parameter is that specified by KIND; otherwise, kind type parameter is that of X.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if argument is empty.

18.4 ASIN(X)

Description. Interval enclosure of the inverse sine intrinsic over an interval.

$$\text{asin}(X) \supseteq \ \{\{[\text{asin}(x)]\} \mid x \in X \cap \ [-1, 1]\}.$$

Special values. $\text{asin}(\mathbb{R}^*) = [-\pi/2, \pi/2]$.

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

18.5 ATAN(X)

Description. Interval enclosure of the inverse tangent intrinsic over a pair of intervals.

$$\text{atan}(X) \supseteq \ \{\{[\text{atan}(x)]\} \mid x \in X\}.$$

Special values. $\text{atan}([+\infty]) = [\pi/2]$, $\text{atan}([-\infty]) = [-\pi/2]$, $\text{atan}(\mathbb{R}^*) = [-\pi/, \pi/2]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

18.6 ATAN2(Y,X)

Description. Interval enclosure of the inverse tangent intrinsic over a pair of intervals.

$\text{atan2}(Y, X) \supseteq \ \{\{[\text{atan2}(y, x)]\} \ y \in Y, \mid x \in X\}$, where

$\text{atan2}(y, x) = \theta$, given $h \sin\theta = y, h\cos\theta = x$ and $h = (x^2 + y^2)^{1/2}$

Class. Elemental function.

Special values. The following table from [20] contains all the indeterminate forms of the `ATAN2` intrinsic.

y	x	$\{[\sin\theta]\}$	$\{[\cos\theta]\}$	$\{[\theta]\}$
0	0	$[-1,1]$	$[-1,1]$	$[-\pi,\pi]$
$+\infty$	$+\infty$	$[0,1]$	$[0,1]$	$\left[0,\frac{\pi}{2}\right]$
$+\infty$	$-\infty$	$[0,1]$	$[-1,0]$	$\left[\frac{\pi}{2},\pi\right]$
$-\infty$	$-\infty$	$[-1,0]$	$[-1,0]$	$\left[-\pi,\frac{-\pi}{2}\right]$
$-\infty$	$+\infty$	$[-1,0]$	$[0,1]$	$\left[\frac{-\pi}{2},0\right]$

Arguments. Y is of type interval, X is of the same type and kind type parameter as Y.

Result characteristics. Same as the arguments.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if one or both arguments are empty.

To get a sharp interval enclosure (denoted by Θ), in the case when $\overline{x} < 0$ and $0 \in Y$, the following convention is used to uniquely define all possible returned interval angles (see [19] for complete discussion):

$$-\pi \quad < \quad m(\Theta) \leq \pi. \tag{20}$$

This choice, together with

$$0 \leq w(\Theta) < 2\pi. \tag{21}$$

results in a unique definition, of the interval angles, Θ, that atan2(y, x) must include.

The following table contains the tests and the arguments of the real `ATAN2` function that can be used to compute the endpoints of Θ in an algorithm that satisfies the constraints in (20) and (21). The first two columns define the cases to be distinguished. The third column contains the range of possible values of $m(\Theta)$. The last two columns show how the endpoints of Θ are computed, using the real `ATAN2` intrinsic function. Of course, directed rounding must be employed to guarantee containment.

Y	X	$m(\Theta)$	$\underline{\theta}$	$\overline{\theta}$
$-\underline{y} < \overline{y}$	$\overline{x} < 0$	$\frac{\pi}{2} < m(\Theta) < \pi$	ATAN2 $(\overline{y}, \overline{x})$	ATAN2 $\left(\underline{y}, \overline{x}\right) + 2\pi$
$-\underline{y} = \overline{y}$	$\overline{x} < 0$	$m(\Theta) = \pi$	ATAN2 $(\overline{y}, \overline{x})$	$2\pi - \underline{\theta}$
$\overline{y} < -\underline{y}$	$\overline{x} < 0$	$-\pi < m(\Theta) < -\frac{\pi}{2}$	ATAN2 $(\overline{y}, \overline{x}) - 2\pi$	ATAN2 $\left(\underline{y}, \overline{x}\right)$

18.7 CEILING(X [, KIND])

Description. Returns the least integer greater than or equal to the supremum of the interval.

Class. Elemental function.

Arguments. X is of type interval

KIND (optional) is a scalar integer initialization expression.

Result characteristics. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, kind type parameter is that of default integer type.

Result value. The result value is `CEILING( SUP (X))`.

`CEILING (∅) = CEILING(NaN), CEILING(` $\mathbb{R}^*$`) = + MAX_INT`

18.8 COS(X)

Description. Interval enclosure of the cosine intrinsic over an interval[27]

$$\cos(X) \supseteq \{\{[\cos(x)]\} \mid x \in X\}.$$

Special values. $\cos(\mathbb{R}^*) = [-1, 1]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

Examples.

```
COS([ -INF, +INF])= [ -1,1]
```

[27] This is a quality of implementation option to provide a sharp enclosure of denegerate interval angles modulo π.

18.9 COSH(X)

Description. Interval enclosure of the hyperbolic cosine intrinsic over an interval.

$$\cosh(X) \supseteq \ \{\{[\cosh(x)]\} \mid x \in X\}.$$

Special values. $\cosh([+\infty]) = [+\infty]$, $\cosh([-\infty]) = [+\infty]$.

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

18.10 EXP(X)

Description. Interval enclosure of the exponential intrinsic over an interval.

$$\exp(X) \supseteq \ \{\{[\exp(x)]\} \mid x \in X\}.$$

Special values. $\exp(\ \mathbb{R}^*) = [0, +\infty]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if argument is empty.

Examples.

```
EXP([ -INF, +INF])= [ 0, +INF]
```

18.11 FLOOR(X [, KIND])

Description. Returns the greatest integer less than or equal to the infimum of the interval.

Class. Elemental function.

Arguments. X is of type interval

KIND (optional) is a scalar integer initialization expression.

Result characteristics. Integer. If KIND is present, the kind type parameter is that specified by KIND; otherwise, kind type parameter is that of default integer type.

Result value. The result value is `FLOOR( INF (X))`.

`FLOOR (∅) = FLOOR(NaN), FLOOR(` $\mathbb{R}^*$`) = - MAX_INT`

18.12 LOG(X)

Description. Interval enclosure of the natural logarithm intrinsic over an interval.

$$\log(X) \supseteq \ \{\{[\log(x)]\} \mid x \in X \cap \ [0, +\infty]\}.$$

Special values. $\log([0]) = [-\infty] \subset [-\infty, -\overline{fp}]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

Examples.

`LOG([0])= [ -INF, -` $\overline{fp}$`]`

18.13 LOG10(X)

Description. Interval enclosure of the common logarithm intrinsic over an interval.

$$\log 10(X) \supseteq \ \{\{[\log 10(x)]\} \mid x \in X \cap \ [0, +\infty]\}.$$

Special values. $\log 10([0]) = [-\infty] \subset [-\infty, -\overline{fp}]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

Examples.

`LOG10([0])= [ -INF, -` $\overline{fp}$`]`

18.14 MOD(X, Y)

Description. Interval enclosure of the modulo intrinsic over an interval.

$$\mathrm{mod}(X,Y) \supseteq \ \{\{[\mathrm{mod}(x,y)]\} \mid x \in X,\ y \in Y\},\ \text{where } \mathrm{mod}(x,y) = x - y * INT(x/y).$$

Special values. If $0 \in Y$ then $\mathrm{mod}(X,Y) = \ \mathbb{R}^*$

Class. Elemental function.

Arguments. The arguments are of type interval and all have the same type and kind type parameter.

Result characteristics. Same as X.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if one or both arguments are empty.

Examples.

```
MOD( [0], [0] ) = [-INF, +INF ]
```

18.15 PRECISION(X)

Description. Returns the decimal precision of the model representing real numbers with the same kind type parameter as the argument.

Arguments. X is of type interval. It may be scalar or array valued.

Result characteristics. Default integer scalar.

Result value. `PRECISION(INF(X))`.

18.16 RANGE(X)

Description. Returns the decimal exponent range of the model representing real numbers with the same kind type parameter as the argument.

Arguments. X is of type interval. It may be scalar or array valued.

Result characteristics. Default integer scalar.

Result value. `RANGE(INF(X))`.

18.17 SIGN(X, Y)

Description. Interval enclosure of the `SIGN` intrinsic over an interval.

$$\texttt{SIGN}(X, Y) \supseteq \ \{\{[\texttt{SIGN}(x, y)]\} \mid x \in X, y \in Y\}.$$

Special values. $\forall X$, `SIGN(` $\mathbb{R}^*$, X)= $\mathbb{R}^*$

Class. Elemental function.

Arguments. The arguments are of type interval.

Result characteristics. Same as X.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if one or both arguments are empty.

Examples:

```
SIGN([-2,-1],[3,4]) = [1,2]
SIGN([-1,2],[-4,-3]) = [-2,0]
SIGN([1,2],[-4,-3]) = [-2,-1]
SIGN([1,2],[0,1]) = [1,2]
SIGN([1,2],[-1,0]) = [-2,2]
SIGN([1,2],[-INF,+INF]) = [-2,2]
SIGN([-INF,+INF],[3,4]) = [-INF,+INF]
```

18.18 SIN(X)

Description. Interval enclosure of the sine intrinsic over an interval [28].

$$\sin(X) \supseteq \ \{\{[\sin(x)]\} \mid x \in X\}.$$

[28] It is a quality of implementation option to provide a sharp enclosure of denegerate interval angles modulo 2π.

Special values. $\sin(\ \mathbb{R}^*) = [-1, 1]$

Arguments. X is of type interval

Result characteristics. Same as the arguments.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

Examples.

```
SIN([ -INF, +INF])= [ -1,1]
```

18.19 SINH(X)

Description. Interval enclosure of the hyperbolic sine intrinsic over an interval.

$$\sinh(X) \supseteq \ \{\{[\sinh(x)]\} \mid x \in X\}.$$

Special values. $\sinh([+\infty]) = [+\infty]$, $\sinh([-\infty]) = [-\infty]$.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if the argument is empty.

18.20 SQRT(X)

Description. Interval enclosure of the square root intrinsic over an interval.

$$\sqrt{X} \supseteq \ \{\{[\exp(\frac{1}{2}\ln(x))]\} \mid x \in X \cap \ [0, +\infty]\}.$$

Special values. $\exp(\frac{1}{2}\ln(\mathbb{R}^*)) = [0, +\infty]$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description. The result is empty if argument is empty.

Examples.

`SQRT([ -INF, x])= [EMPTY]` if $x < 0$

`SQRT([ -INF, +INF])= [ 0, +INF]`

18.21 TAN(X)

Description. Interval enclosure of the tangent intrinsic over an interval[29].

$$\tan(X) \supseteq \{\{[\tan(x)]\} \mid x \in X\}.$$

Special values. $\tan([\pi/2 + k\pi]) = \{-\infty\} \cup \{+\infty\} \subset [-\infty, +\infty], k \in \mathbb{Z}$[30] ,
$\tan([-\infty,]) = \mathbb{R}^*$
$\tan([+\infty]) = \mathbb{R}^*$
$\tan(\mathbb{R}^*) = \mathbb{R}^*$

Class. Elemental function.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if argument is empty.

18.22 TANH(X)

Description. Interval enclosure of the hyperbolic tangent intrinsic over an interval.

$$\tanh(X) \supseteq \{\{[\tanh(x)]\} \mid x \in X\}.$$

Special values. $\tanh([+\infty]) = [1]$, $\tanh([-\infty]) = [-1]$, $\tanh([\mathbb{R}^*) = [-1, 1]$.

Class. Elemental function.

[29] It is a quality of implementation option to provide a sharp enclosure of denegerate interval angles modulo 2π.

[30] The set of integers.

Arguments. X is of type interval

Result characteristics. Same as the argument.

Result value. The interval result value is an enclosure for the specified interval. An ideal enclosure is a fp-interval of minimum width that contains the exact mathematical interval in the description.

The result is empty if argument is empty.

19 Conversions to interval types

Explicit conversions to the interval from integers, reals or intervals of different types are performed with type conversion intrinsics `INTERVAL, SINTERVAL, DINTERVAL` and `QINTERVAL.` [31]

19.1 `INTERVAL(X [, Y, KIND])`

Description. Convert to interval type

Class. Elemental function.

Arguments.

`X` is of type integer, real, or interval. `Y` (optional) is of type integer or real. If `X` is of type interval, `Y` must not be present.

KIND (optional) is a scalar integer initialization expression.

Result characteristics. The result is of type interval. If KIND is present, the kind type parameter is that specified by KIND; otherwise, the kind type parameter is that of the default interval type.

Containment is guaranteed if X is of type interval For example given `INTERVAL(16) X` the result of `INTERVAL(X, KIND=4)` contains interval `X`

Containment is **not guaranteed** otherwise.

For example, given `REAL(8) X,Y` the result of `INTERVAL(X,Y, KIND=4)` does not necessarily contain the internal interval [X,Y]

In comparison with interval literal constants the result of (for example)`INTERVAL(1.1,1.3)` does not necessarily contain the mathematical interval [1.1, 1.3], whereas `[1.1,1.3]` does contain [1.1, 1.3].

Result value. The interval result value is a valid floating–point interval (see section 3).

[31] These intrinsics insulate their arguments from widest–need interval expression evaluation, see section 9

If Y is absent and X is an interval, then INTERVAL(X [,KIND]) is an interval containing X:

INTERVAL(X [,KIND]) is an interval with left and right endpoints [XL, XU], where XL = REAL(INF(X) [,KIND]) rounded down, so that XL .LE. INF(X) and XU = REAL(SUP(X)[,KIND]) rounded up, so that XU .GE. SUP(X).

If both X and Y are present then INTERVAL(X,Y [,KIND]) is an interval with left and right endpoints equal to REAL(X [,KIND]) and REAL(Y [,KIND]) respectively. [32]

If Y is absent and X is not an interval, then INTERVAL(X [, KIND]) is INTERVAL(X,X [, KIND]).

If both X and Y are present and Y is less than X or if either X or Y or both do not represent a mathematical integer or real number (e.g. one or both real argument is a NaN) then $\mathbb{R}^*$ is returned.

19.2 DINTERVAL(X, [Y])

Description. Convert to INTERVAL(8) type.

DINTERVAL(X, [Y]) is equivalent to INTERVAL(X, [Y], KIND=8)

Class. Elemental function.

19.3 SINTERVAL(X, [Y])

Description. Convert to INTERVAL(4) type.

SINTERVAL(X, [Y]) is equivalent to INTERVAL(X, [Y], KIND=4)

Class. Elemental function.

19.4 QINTERVAL(X, [Y])

Description. Convert to INTERVAL(16) type.

QINTERVAL(X, [Y]) is equivalent to INTERVAL(X, [Y], KIND=16)

Class. Elemental function.

[32] Because interval [inf,inf] is not supported in the current release interval [max_real,inf] has to be returned instead of it.

19.5 Conversion examples

```
REAL R, S, T
REAL(8) R8, T81, T82
INTERVAL(4) X, Y
INTERVAL(8) DX
                                    Interpretation
X = INTERVAL(0.1, 0.1, KIND=4)      ! R=0.1 ; X=INTERVAL(R,R, KIND=4)
X = INTERVAL(0.1, KIND=4)           ! R=0.1 ; X=INTERVAL(R,R, KIND=4)
DX = DINTERVAL(R+S, T+R8)           ! T81=R+S; T82=S+R8; DX = DINTERVAL(T81,T82)
DX = INTERVAL(Y, KIND=8)            ! Converts Y to a containing INTERVAL(8) interval

! Produce a sharp interval containing a given real number
DX = INTERVAL(-TINY(R8),TINY(R8), KIND=KIND(R8)) + INTERVAL(R8,KIND=KIND(R8))


! Some code setting both  R and S  to zero

T = R/S   ! T is 0/0, which creates a NaN

! Produce interval [-INF, INF]
X = DINTERVAL(T, S)
Y = DINTERVAL(T, T)
```

20 Interval array intrinsics

Interval versions of the following array intrinsics are supported:
ALLOCATED(),
ASSOCIATED(),
CSHIFT(),
DOT_PRODUCT(),
EOSHIFT(),
KIND(),
LBOUND(),
MATMUL(),
MAXVAL()
MERGE(),
MINVAL(),
NULL(),

PACK(),
PRODUCT(),
RESHAPE(),
SHAPE(),
SIZE(),
SPREAD(),
SUM(),
TRANSPOSE(),
UBOUND(),
UNPACK().

The MINLOC(), and MAXLOC() intrinsics are not defined for interval arrays because the MINVAL and MAXVAL intrinsic applied to an interval array returns an interval that is not an element of the array[33]. For example MINVAL((/[1,2],[3,4]/)) = [1,3] and MAXVAL((/[1,2],[3,4] = [2,4].

Array versions of the following interval specific intrinsics are supported:
INF(),
SUP(),
ABS(),
MAG(),
MAX(),
MID(),
MIG(),
MIN(),
NDIGITS(),
WID(),
INT().

Array versions of the following interval mathematical intrinsics are supported:
ACOS(),
AINT(),
ANINT(),
ASIN(),
ATAN(),
ATAN2(),
CEILING(),
COS(),
COSH(),
EXP(),
FLOOR(),
LOG(),

[33]See description of the MIN intrinsic.

LOG10(),
MOD(),
SIN(),
SIGN(),
SINH(),
SQRT(),
TAN(),
TANH().

Array versions of the following interval constructors are supported:
INTERVAL(),
DINTERVAL(),
SINTERVAL(),
QINTERVAL().

21 Interval I/O editing

21.1 Interval input

Interval input maps a decimal interval into its internal floating–point binary representation

The internal fp–interval must contain the external interval regardless of the number of digits in the input field or the precision of the internal representation.

Input of an invalid interval with right endpoint less than left endpoint is an I/O error and must cause the program to abort, unless alternative action (ERR clause) has been provided specifically in the program. In this case $\mathbb{R}^*$ is returned.

21.2 Interval output

Interval output maps the internal binary representation of an interval into an external decimal representation.

The external interval must contain the internal fp–interval regardless of the number of digits in the field or the precision of the internal representation.

21.3 External interval representation

Let x_d be a decimal number reprsented by a *d-input-field*, this is an input format corresponding to the D edit descriptor)

The *interval-input-field* can have one of the three forms:

- "x_d" – represents a non-degenerate mathematical interval

  $x_d + [-1, 1]_{\mathbf{uld}}$ (unit in the last digit)·

  Thus, trailing zeros are significant and `0.10000000` represents interval `[0.099999999,0.100000001]`, whereas `0.10` represents interval `[0.09,0.11]`. Note: `100E-1` represents interval `[9.9,10.1]` since all trailing zero digits in the mantissa are significant.

- "$[x_d]$" – represents a degenerate mathematical interval $[x_d, x_d]$.

- "$[x_{d_1}, x_{d_2}]$" – represents a mathematical interval $[x_{d_1}, x_{d_2}]$, with $x_{d_1} \leq x_{d_2}$.

On input, if either endpoint is not exactly representable, the left endpoint is rounded down and the right endpoint is rounded up to numbers known to contain the exact decimal value.

A degenerate interval [1.5, 1.5] can be externally represented in any of the following ways:

`[1.5E0]`, `[1.5]`, `[1.5, 1.5]`, `[0.15E1, 1.5]`.

21.4 Interval edit descriptors

The `VF`$w.d$, `VE`$w.d$`E`e, `VEN`$w.d$`E`e, `VES`$w.d$`E`e, `VG`$w.d$`E`e, `Y`$w.d$`E`e, `F`$w.d$, `E`$w.d$`E`e, or `G`$w.d$`E`e edit descriptors may be used to specify the input/output formatting of interval data. w and d specifiers must not be omitted.

During input:

- All interval edit descriptor have the same semantics. The parameter w, specifies the field width containing external data.

- If the input field contains a decimal point, the specification of d is ignored. If the decimal point is omitted, the d designates the position of the "imaginary" decimal point of the input value; that is, the input value is multiplied by 10^{-d}.

- Leading blanks are not significant.

- Because trailing zeros are significant in interval input `BZ` control edit descriptor does not affect the interpretation of leading and trailing blanks when `Y`, `VF`, `VE`, `VEN`, `VES`, `VG` and `F`,`E`,`G` edit descriptors are applied to intervals.

- If an input list item corresponding to the `A`, `Z`, `O` or `I` edit descriptor is of interval type, $\mathbb{R}^*$ is returned.

During output:

- If the output field's width, w, in `VF, VE, VEN, VES,` or `VG` edit descriptor is even, then the field is filled with one leading blank character and $w-1$ is used for the output field's width.
- The `VF, VE, VEN, VES,` and `VG` edit descriptors provide $[inf, sup]$ – style formatting of intervals, using `F, E, EN, ES, G` edit descriptors, respectively, for the interval endpoints. The `Y` edit descriptor is used for single–number interval output, see section 21.10.
- The `F, E, G` edit descriptors applied to intervals have the same meaning as the `Y` edit descriptor except that if the `F` or `G` specifier is used, the output field may have the form prescribed by the `F` edit descriptor, (see section 21.10). If the `E, EN, ES` specifiers are used, the output field may only have the form prescribed by the `E, EN, ES` edit descriptor.
- If an output list item corresponding to the `VF, VE, VEN, VES, VG` or `Y` edit descriptor is any type other than interval type the whole output field is filled with asterisks.
- If an output list item corresponding to the `A, Z, O` or `I` edit descriptor is of interval type the whole output field is filled with asterisks.
- A positive interval endpoint may be prefixed with a plus sign; a negative endpoint is always prefixed with minus sign; and the zero interval endpoint is never prefixed with a leading plus or minus.

During input and output

- The `P` edit descriptor may be used to change the scale factor for the `Y, VF, VE, VEN, VES` or `VG` descriptors and for `F,E` or `G` edit descriptors when applied to intervals.
- The empty interval is input/output as a case insensitive string "`EMPTY`" enclosed in "`[`" and "`]`". The string "`EMPTY`" may be preceded or followed by blanks.
- A minus or plus infinite interval endpoint is input/output as a case insensitive string "`INF`" prefixed with a minus or an optional plus sign.

21.5 Interval `VF` editing

An output value in a `VF`$w.d$ edit field has the form

[*f-output-field*$_1$, *f-output-field*$_2$], where $w' = (w-3)/2$ and *f-output-field* is the output field for the `F`$w'.d$ format (for non–interval types),

w' must allow for a minus sign, at least one digit to the left of the decimal point, the decimal point, and d digits to the right of the decimal point. Therefore $w' \geq d + 3$. Additionally w must allow for "`[`", "`]`" and ",". Therefore $w \geq 2(d + 3) + 3$ or equivalently w must allow for two fields in `F`w'.d format and and "`[`", "`]`" and ",". Therefore $w \geq 2 * w' + 3$

Using the specified format, if one or both *f-output-field*$_1$ or *f-output-field*$_2$ fields are too small for the output of the corresponding endpoint, then only that field must be filled with asterisks.

For both input and output, the symbols "`[ `", "`] `" and "," that are part of the field are counted as part of the width w of the overall `VF` field. The total width of the field is thus $2w' + 3$.

Input of the degenerate interval `[1.5]` and the output of resulting internal interval with the `VF18.5` descriptor has the following form:

```
123456789012345678 ! position number
 [1.50000,1.50000]
```

21.6 Interval `VE` editing

An output value in a `VE`w.d`[E`e`]` edit field has the form

[*e-output-field*$_1$, *e-output-field*$_2$], where $w' = (w - 3)/2$ and *e-output-field* is the output field for the `E`w'.d`[E`e`]` format (for real types).

w' need not allow for a minus sign, but must allow for a zero, the decimal point, and d digits to the right of the decimal point, and an exponent. Therefore, for nonnegative numbers, $w' \geq d + 6$; if e is present, then $w' \geq d + e + 4$. For negative numbers $w' \geq d + 7$; if e is present, then $w' \geq d + e + 5$, and w must allow for two fields using the `E`w'.d`[E`e`]` format and additionally for "`[`", "`]`" and ",". Therefore $w \geq 2w' + 3$

Using the specified format, if one or both *e-output-field*$_1$ or *e-output-field*$_2$ fields are too small for the output of the corresponding endpoint, then only that field must be filled with asterisks.

For both input and output, the symbols "`[ `", "`] `" and "," that are part of the field are counted as part of the width w of the overall `VE` field. The total width of the field is thus $2w' + 3$.

Example: Let the interval variable, X be defined in a program, and suppose the program containes the statement :

```
WRITE(*, '(1X, VE27.5E1)') X
```

Let the internal representation of `X` be:

```
[1.9921875, 2.9921875]
```

Then valid output produced by the WRITE statement is

```
12345678901234567890123456 ! position number
[  0.19921E+1,  0.29922E+1]
```

21.7 Interval VEN editing

The VEN edit descriptor may be used to describe the editing of interval data. It differs from the VE edit descriptor only in the form of the output field.

An output value in a VEN$w.d$[Ee] edit field has the form

[*en-output-field*$_1$, *en-output-field*$_2$], where $w' = (w - 3)/2$ and *en-output-field* is the output field for the EN$w'.d$[Ee] format (for real types).

21.8 Interval VES editing

The VES edit descriptor may be used to describe the editing of interval data. It differs from the VE edit descriptor only in the form of the output field.

An output value in a VES$w.d$[Ee] edit field has the form

[*es-output-field*$_1$, *es-output-field*$_2$], where $w' = (w - 3)/2$ and *es-output-field* is the output field for the ES$w'.d$[Ee] format (for real types).

21.9 Interval VG editing

For interval input, the interval VG edit descriptor is identical to the VF edit descriptor. For interval output with VG$w.d$ or VG$w.d$Ee, two decimal numbers are printed, preceded by "[", separated by ", " and followed by "] ". The formats of the lower bound and upper bound are determined in accordance with the rules for non–interval G$w.d$ or G$w.d$Ee editing for the lower and upper bound of the output interval, respectively. The printed lower and upper bounds must contain the internal interval representation.

Using the specified format, if one or both output fields are too small for the output of the corresponding endpoint, then only that field must be filled with asterisks.

21.10 Single number interval Y editing

The Y edit descriptor is used to output an interval in the single number interval representation (see also [17]) with implicit bounds of ± 1 unit in the last exhibited decimal digit.

The letter **Y** is chosen as a visual indicator of the relationship of the components of an interval value to the single number used to represent that value, as illustrated here:

inf sup
Y
single number
representation

The general form of the `Y` edit descriptor is: `Y`w.d`[E`e`]`

The d specifier sets the number of places allocated for displaying significant digits[34].

The e specifier (if present) defines the number of digits displayed in the exponent part of the output field. The presence of the e specifier forces the output field to have the form prescribed by the `E` (as opposed to `F`) edit descriptor.

In certain cases the single–number interval representation is less informative than a $[inf, sup]$ representation of the interval `X`. This is the case when either the interval itself contains zero or infinity, or the interval's single-number form represents an interval containing zero.

In these cases, to produce sharper external representation of the internal fp–interval, `VG`w.d'`E`e is used with d' allowing for the maximal number of significant digits to be output[35].

For example: The single-number interval representation of the interval `[-1, 10]` is `0.E1` which is the much wider interval `[-10, 10]` therefore the output

```
12345678901234567890! position number
 [ -1.     , 0.1E+02]
```

may be produced with `Y20.`d

The single-number interval representation of the interval `[1,6]` is `1.E1` which is the much wider interval `[0, 20]` . Therefore the output

```
12345678901234567890! position number
 [  1.     ,  6.     ]
```

may be produced with `Y20.`d

If there is sufficient field width, the `E` or `F` style will be used. The `E` or `F` descriptor is used that can display the greater number of digits of the interval in the output list. If the number of digits displayed using the `E` or `F` descriptor is the same, the `F` descriptor is used.

[34]The actual number of displayed significant digits may be more or less than d
[35]Quality of implementation opportunity.

The d specifier sets the number of significant digits displayed. It is possible to have more significant digits using the `F` descriptor. Positions corresponding to the digits that are not displayed are filled with blanks.

An additional exponent digit can be displayed instead of the "`E`" character if necessary.

The exponent field (if present) is always right justified in the output field.

w must allow for a minus sign, "`[`", "`]`", at least one digit to the left of the decimal point, the decimal point, d digits to the right of the decimal point, and an exponent. Therefore $w \geq d + 10$. If e is present, $w \geq d + e + 7$.

The decimal point is located in position $p = e + d + 4$, counting from the right of the field of w characters.

If $w \geq p + 2$ (2 positions for left bracket and sign), there will be character positions available for the integer part of a number in F format. If there are not enough character positions available to display a number using the `F` format and keep the decimal point located in the p–th position, then the `E` format is used.

If it is possible to represent a degenerate interval within the field width, the output field is enclosed in obligatory "`[`" and "`]`" and the
`F`$(w-2).(d+e+2)$, the `F`$(w-2).(d+3+2)$ or the `E`$(w-2).d$`E`e specifier is used.

Otherwise the form of the output field is:

y-output-field

where *y-output-field* has the form specified by either the

`F`$(w-2).(d+e+2)$, the `F`$(w-2).(d+3+2)$, or the `E`$(w-2).d$`E`e specifier.

The following details of the output field are processor–dependent:

- the leading zero before the decimal point;
- the plus sign for a positive value; and,
- the form of the exponent if the absolute value of the exponent is ≤ 999 or > 9999 and "`E`" character is omitted.

If the width of the output field allows, the leading zero before the decimal point should be output. If the width of the output and exponent fields allow, the form of the exponent field should be "`E`", followed by a sign, followed by three or e digits.

Thus the following output is desirable:

```
FORMAT(Y17.6)
12345678901234567! position number
  -0.123456789    ! represents the interval [-0.123456790, -0.123456788]
  -0.12345678912  ! represents the interval [-0.123456789124 ,-0.123456789122]
  -1.2            ! represents the interval [-1.3,-1.1]
[ -1.50000000000] ! represents the  degenerate interval [-1.5,-1.5]

FORMAT(Y17.6E4 )
12345678901234567  ! position number
 -0.123457E+0000  ! represents the interval [-0.123456790, -0.123456788]
[ 0.123000E+0021]  ! represents the interval [0.123E21, 0.123E21]
  0.123   E+0021  ! represents the interval [0.122E21, 0.124E21]
 -0.12    E+0305  ! represents the interval [-0.13E+0305, -0.11E+0305]
  0.12    E+1234  ! represents the interval [0.11Q+1234,0.13Q+1234]
123456789012345  ! position number

FORMAT(Y15.5E3 )
***************    ! Result of trying to output .12Q1234 using Y15.5E3

FORMAT(Y15.5 )
123456789012345  ! position number
  0.123  E+021  ! represents the interval [0.122E+021, 0.124E+021]
 -1.2           ! represents the interval [-1.3,-1.1]
 -0.120  E+003  ! represents the interval [-0.121E+003,-0.119E+003]
 -0.12   E+305  ! represents the interval [-0.13E+305,-0.11E+305]
  0.12   E+305  ! represents the interval [0.11E+305, 0.13E+305]
  0.12   +1234  ! represents the interval [0.11Q+1234,0.13Q+1234]
1234567890123  ! position number
```

21.10.1 Single number I/O and internal base conversions

In single number interval I/O, input immediately followed by output can appear to suggest that a decimal digit of accuracy has been lost, when in fact radix conversion has caused a 1 or 2 ulp (unit in the last place) increase in the width of the stored input interval. For example, an input of `1.37` followed by an immediate print may result in `1.3` being output.

Users must use character input and output to exactly echo input values and internal reads to convert input values into fp–intervals.

For example, an interaction with the program

```
      INTERVAL X
```

```
      CHARACTER*30 BUFFER
C     TEST INTERNAL INPUT AND OUTPUT
      PRINT  *, 'ENTER INPUT INTERVAL:'
      READ   (*,'(A12)'), BUFFER
      PRINT  *, 'YOUR INPUT WAS:', BUFFER
      READ(BUFFER, '(Y12)'), X
      PRINT  *, 'RESULT INTERVAL  IS:', X
      PRINT  *, 'SINGLE NUMBER INTERVAL OUTPUT  IS:'
      PRINT( '(Y12.2)'), X
      END
```

results in

```
 ENTER INPUT INTERVAL:
1.37
 YOUR INPUT WAS:1.37
 RESULT INTERVAL  IS: [1.35999999999999987E+00, 1.38000000000000012E+00]
 SINGLE NUMBER INTERVAL OUTPUT  IS:
1.3
```

21.11 List–directed interval I/O

21.11.1 List–directed interval input

If an input list item is of interval type the input record must contain an interval or a null value.

An input interval value may have the same form as an interval, real or integer literal constant. If an interval value has the form of a real or integer literal constant, (x_d without "[" and "]"), the input is interpreted as a non-degenerate mathematical interval

$x_d + [-1, 1]_{\mathbf{uld}}$ (unit in the last digit)

If $[inf, sup]$ input style is used, the end of record may occur between the left endpoint and the comma or between the comma and the right endpoint.

A null value, such as between two consecutive commas, means that the corresponding variable in the input list is not changed. A null value must not be used for either left or right endpoints of an interval constant, but a single null value may be used to represent a complete interval datum.

21.11.2 List–directed interval output

Real constants for left and right endpoints are produced with the effect of either an `F` edit descriptor or an `E` edit descriptor, depending on the magnitude x of the value and a range $10^{d_1} \leq x \leq 10^{d_2}$, where d_1 and d_2 are processor dependent integers, ($d_1 = -2$ and $d_2 = +8$ in Sun f95). If the magnitude x is within this range, the endpoint is produced using `0PF`$w.d$; ohterwise `1PE`$w.d$`E`e is used.

In Sun f95 the values for d and e are the same as for the real types corresponding to the interval endpoints. The value of w reflects the need to conveniently accomodate two real values of corresponding types and three addtional characters for "`[`", "`]`" and the comma.

The end of record may occur between the comma and the right endpoint only if the entire constant is as long as, or longer than, an entire record. The only embedded blanks permitted within an interval constant are between the comma and the end of the record and one blank at the beginning of the record.

21.12 Namelist interval I/O

Interval values can be input and output using NAMELIST I/O.

21.13 File compatibility with future releases

The following table shows how (in the "Simple" system, specified herein) interval internal binary representations are converted to the external binary and external character representations (`"x"` stands for an external character representation of `x`). In this table, $x \neq$ `inf` and $x \neq$ `0`.

External binary	Output	Internal binary	Output	External character
`[-0,+0]`	⟵	`[-0,-0]`	⟶	`[0,0]`
`[-0,+0]`	⟵	`[-0,+0]`	⟶	`[0,0]`
`[-0,+0]`	⟵	`[+0,-0]`	⟶	`[0,0]`
`[-0,+0]`	⟵	`[+0,+0]`	⟶	`[0,0]`
`[-0,x]`	⟵	`[+0,x]`	⟶	`[0,"x"]`
`[-0,x]`	⟵	`[-0,x]`	⟶	`[0,"x"]`
`[x, +0]`	⟵	`[x, -0]`	⟶	`["x",0]`
`[x, +0]`	⟵	`[x, +0]`	⟶	`["x",0]`
`[x,-inf]`	⟵	`[x, +inf]`	⟶	`["x", +INF]`
`[-0,-inf]`	⟵	`[+0, +inf]`	⟶	`[0,+INF]`
`[+inf, x]`	⟵	`[-inf,x]`	⟶	`[-INF,"x"]`
`[+inf,+0]`	⟵	`[-inf, +0]`	⟶	`[-INF,0]`

The following table shows how (in the "Simple" interval system) external binary and character strings are converted into the internal binary representation ("x" stands for an external character representation of x). In this table, $x \neq$ inf, $x \neq$ 0.

External binary	Input	Internal binary	Input	External character
[-0,x]	$\longrightarrow$	[-0,x]	$\longleftarrow$	[0,x]
[+0,x]	$\longrightarrow$	[+0,x]	$\longleftarrow$	[1e-5000, "x"]
[x, +0]	$\longrightarrow$	[x,+0]	$\longleftarrow$	["x",0]
[x, -0]	$\longrightarrow$	[x,-0]	$\longleftarrow$	["x",-1E-5000]
[x,-inf]	$\longrightarrow$	[x,+inf]	$\longleftarrow$	["x", +INF]
[x,+inf]	$\longrightarrow$	[x,+inf]	$\longleftarrow$	["x",+INF]
[-0,-inf]	$\longrightarrow$	[-0,+inf]	$\longleftarrow$	[0, +INF]
[-0,+0]	$\longrightarrow$	[-0,+0]	$\longleftarrow$	[0,0]
[+inf,+0]	$\longrightarrow$	[-inf,+0]	$\longleftarrow$	[-INF,0]
[+inf,-0]	$\longrightarrow$	[-inf,-0]	$\longleftarrow$	[-INF, -1E-5000]
[+inf,x]	$\longrightarrow$	[-inf,x]	$\longleftarrow$	[-INF,"x"]
[+inf, $-\underline{fp}$]	$\longrightarrow$	[-inf, $-\underline{fp}$]	$\longleftarrow$	[-INF, -INF]
[-0, $\underline{fp}$]	$\longrightarrow$	[-0, $\underline{fp}$]	$\longleftarrow$	[0,1E-5000]

Note: $\underline{fp}$ – is the smallest positive representable floating–point number.

The following table shows how (in the "Sharp" and "Full" interval systems) character strings representing intervals with endpoints resulting from floating–point underflow or overflow are converted into the internal "Simple" binary representation ("x" stands for an external character representation of x). The results are included in this specification to guarantee binary file compaibility between "Simple", "Sharp" and "Full" interval systems. In this table, $x \neq$ inf, $x \neq$ 0, "(0" represents underflow and "+INF)" represents overflow.

Internal binary	Input	External character
[+0,x]	$\longleftarrow$	(0,"x"]
[x,−0]	$\longleftarrow$	["x",0)
[x,+inf]	$\longleftarrow$	["x", +INF)
[+0,+inf]	$\longleftarrow$	(0, +INF)
[-inf,−0]	$\longleftarrow$	(-INF,0)
[-inf,x]	$\longleftarrow$	(-INF,"x"]

External representations (0,0) , (-INF,-INF) and (+INF,+INF) are invalid in the "Sharp" and "Full" interval systems.

22 Acknowledgments

The authors are very thankful to Keith Bierman, Craig Burley, Eldon Hansen, David Hough, Vladimir Kharchenko, Douglas Priest, Michael Schulte, Wolfgang Walter and Juergen Wolff von Gudenberg for valuable remarks.

23 Appendix A: Interval Intrinsics

Table 1. Interval Arithmetic Intrinsics

Intrinsic Function	Definition	N Args	Generic Name	Specific Name	Argument Type	Function Type
Absolute value	$\mid X \mid$	1	ABS	VDABS VSABS VQABS	INTERVAL(8) INTERVAL(4) INTERVAL(16)	INTERVAL(8) INTERVAL(4) INTERVAL(16)
Truncation	int(X)	1	AINT	VDINT VSINT VQINT	INTERVAL(8) INTERVAL(4) INTERVAL(16)	INTERVAL(8) INTERVAL(4) INTERVAL(16)
Nearest whole number	int(a+.5) a>0 int(a-.5) a<0	1	ANINT	VDNINT VSNINT VQNINT	INTERVAL(8) INTERVAL(4) INTERVAL(16)	INTERVAL(8) INTERVAL(4) INTERVAL(16)
Remainder	A-int(A/B)*B	2	MOD	VDMOD VSMOD	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Transfer of sign	$\mid A \mid$ if B >= 0 $-\mid A \mid$ if B < 0	2	SIGN	VDSIGN VSSIGN	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Choose largest value	max(A,B,...)	>=2	MAX	MAX	INTERVAL	INTERVAL
Choose smallest value	min(A,B,...)	>=2	MIN	MIN	INTERVAL	INTERVAL

Intrinsics MODand SIGN with INTERVAL(16) argument are not supported in the current release.

Table 2. Type Conversion Intrinsics

Conversion to	N Args	Generic Name	Specific Name	Argument Type	Function Type
INTERVAL	1 or 2	INTERVAL		INTERVAL	INTERVAL
				INTERVAL(4)	INTERVAL
				INTERVAL(8)	INTERVAL
				INTEGER	INTERVAL
				REAL	INTERVAL
				DOUBLE	INTERVAL
				QUAD	INTERVAL
INTERVAL(4)	1 or 2	SINTERVAL		INTERVAL	INTERVAL(4)
				INTERVAL(4)	INTERVAL(4)
				INTERVAL(8)	INTERVAL(4)
				INTEGER	INTERVAL(4)
				REAL	INTERVAL(4)
				DOUBLE	INTERVAL(4)
				QUAD	INTERVAL(4)
INTERVAL(8)	1 or 2	DINTERVAL		INTERVAL	INTERVAL(8)
				INTERVAL(4)	INTERVAL(8)
				INTERVAL(8)	INTERVAL(8)
				INTEGER	INTERVAL(8)
				REAL	INTERVAL(8)
				DOUBLE	INTERVAL(8)
				QUAD	INTERVAL(8)
INTERVAL(16)	1 or 2	QINTERVAL		INTERVAL	INTERVAL(16)
				INTERVAL(4)	INTERVAL(16)
				INTERVAL(8)	INTERVAL(16)
				INTERVAL(16)	INTERVAL(16)
				INTEGER	INTERVAL(16)
				REAL	INTERVAL(16)
				DOUBLE	INTERVAL(16)
				QUAD	INTERVAL(16)

Table 3. Trigonometric Intrinsics[36]

Intrinsic Function	Definition	N Args	Generic Name	Specific Name	Argument Type	Function Type
Sine	sin(A)	1	SIN	VDSIN	INTERVAL(8)	INTERVAL(8)
				VSSIN	INTERVAL(4)	INTERVAL(4)
Cosine	cos(A)	1	COS	VDCOS	INTERVAL(8)	INTERVAL(8)
				VSCOS	INTERVAL(4)	INTERVAL(4)
Tangent	tan(A)	1	TAN	VDTAN	INTERVAL(8)	INTERVAL(8)
				VSTAN	INTERVAL(4)	INTERVAL(4)
Arcsine	arcsin(A)	1	ASIN	VDASIN	INTERVAL(8)	INTERVAL(8)
				VSASIN	INTERVAL(4)	INTERVAL(4)
Arccosine	arccos(A)	1	ACOS	VDACOS	INTERVAL(8)	INTERVAL(8)
				VSACOS	INTERVAL(4)	INTERVAL(4)
Arctangent	arctan(A)	1	ATAN	VDATAN	INTERVAL(8)	INTERVAL(8)
				VSATAN	INTERVAL(4)	INTERVAL(4)
	arctan(A/B)	2	ATAN2	VDATAN2	INTERVAL(8)	INTERVAL(8)
				VSATAN2	INTERVAL(4)	INTERVAL(4)

[36]Intrinsics with INTERVAL(16) argument are not supported in the current release

Table 3. Trigonometric Intrinsics[37] (Continued)

Hyperbolic Sine	sinh(A)	1	SINH	VDSINH VSSINH	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Hyperbolic Cosine	cosh(A)	1	COSH	VDCOSH VSCOSH	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Hyperbolic Tangent	tanh(A)	1	TANH	VDTANH VSTANH	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)

Table 4. Other Mathematical Intrinsics[38]

Intrinsic Function	Definition	N Args	Generic Name	Specific Name	Argument Type	Function Type
Square toot	A**(1/2)	1	SQRT	VDSQRT VSSQRT	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Exponential	e**A	1	EXP	VDEXP VSEXP	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Natural logarithm	log(A)	1	LOG	VDLOG VSLOG	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)
Common logarithm	log10(A)	1	LOG10	VDLOG10 VSLOG10	INTERVAL(8) INTERVAL(4)	INTERVAL(8) INTERVAL(4)

[37] Intrinsics with INTERVAL(16) argument are not supported in the current release

[38] Intrinsics with INTERVAL(16) argument are not supported in the current release

Table 5. Interval Intrinsics with REAL, INTEGER or LOGICAL result

Intrinsic Function	Definition	N Args	Generic Name	Specific Name	Argument Type	Function Type
INF	infimum	1	INF	VDINF	INTERVAL(8)	DOUBLE
				VSINF	INTERVAL(4)	REAL(4)
				VQINF	INTERVAL(16)	REAL(16)
SUP	supremum	1	SUP	VDSUP	INTERVAL(8)	DOUBLE
				VSSUP	INTERVAL(4)	REAL(4)
				VQSUP	INTERVAL(16)	REAL(16)
Width	sup-inf	1	WID	VDWID	INTERVAL(8)	DOUBLE
				VSWID	INTERVAL(4)	REAL(4)
				VQWID	INTERVAL(16)	REAL(16)
Midpoint	(inf+sup)/2	1	MID	VDMID	INTERVAL(8)	DOUBLE
				VSMID	INTERVAL(4)	REAL(4)
				VQMID	INTERVAL(16)	REAL(16)
Magnitude	$\max \mid x \mid$ in X	1	MAG	VDMAG	INTERVAL(8)	DOUBLE
				VSMAG	INTERVAL(4)	REAL(4)
				VQMAG	INTERVAL(16)	REAL(16)
Mignitude	$\min \mid x \mid$ in X	1	MIG	VDMIG	INTERVAL(8)	DOUBLE
				VSMIG	INTERVAL(4)	REAL(4)
				VQMIG	INTERVAL(16)	REAL(16)
Test for empty interval	$X = \emptyset$	1	ISEMPTY	VDISEMPTY	INTERVAL(8)	LOGICAL
				VSISEMPTY	INTERVAL(4)	LOGICAL
				VQISEMPTY	INTERVAL(16)	LOGICAL
Floor	floor(X)	1	FLOOR		INTERVAL(8)	INTEGER
					INTERVAL(4)	INTEGER
					INTERVAL(16)	INTEGER
Ceiling	ceiling(X)	1	CEILING		INTERVAL(8)	INTEGER
					INTERVAL(4)	INTEGER
					INTERVAL(16)	INTEGER
Number of digits		1	NDIGITS		INTERVAL	INTEGER
					INTERVAL(4)	INTEGER
					INTERVAL(8)	INTEGER
					INTERVAL(16)	INTEGER

Appendix B

Implementation

Interval addition and subtraction are implemented using IEEE addition and subtraction operations. The negative of infima are computed to save unnecessary rounding mode changes.

Interval Addition

Input: X= [a,b], Y = [c,d]
Output: X+Y
{ Intervals $[-\infty, -\infty]$ and $[+\infty, +\infty]$ cannot occur by construction }
Save_Rounding_Mode ;
Set_Rounding_Mode_Up;
$l = -(-a - c)$
$u = b + d$
Restore_Rounding_Mode ;
return $[l, u]$

Interval Subtraction

Input: X= [a,b], Y = [c,d]
Output: X-Y
{ Intervals $[-\infty, -\infty]$ and $[+\infty, +\infty]$ cannot occur by construction }
Save_Rounding_Mode ;
Set_Rounding_Mode_Up;
$l = -(-a + d)$
$u = b - c$
Restore_Rounding_Mode ;
return $[l, u]$

Interval multiplication and division are implemented using tests to define which pair of endpoints to use when computing infima and suprema. As with addition and subtraction, the negative of infima are computed to save unnecessary rounding mode changes.

Interval Multiplication

Input: X= [a,b], Y = [c,d]

Output: X*Y
Save_Rounding_Mode ;
Set_Rounding_Mode_Up;
if $a > 0$ $\{X > 0\}$
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a \times c)$
    $u = b \times d$
  else if $d < 0$ $\{Y < 0\}$
    $l = -(-b \times c)$
    $u = a \times d$
  else $\{\ 0 \in Y\ \}$
    $l = -(-b \times c)$
    $u = b \times d$
  endif
else if $b < 0$ $\{X < 0\}$
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a \times d)$
    $u = b \times c$
  else if $d < 0$ $\{Y < 0\}$
    $l = -(-b \times d)$
    $u = a \times c$
  else $\{\ 0 \in Y\ \}$
    $l = -(-a \times d)$
    $u = a \times c$
  endif
else $\{\ 0 \in X\}$
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a \times d)$
    $u = b \times d$
  else if $d < 0$ $\{Y < 0\}$
    $l = -(-b \times c)$
    $u = a \times c$
  else $\{\ 0 \in Y\ \}$
    $l = -\max(-a \times d, -b \times c)$
    $u = \max(a \times c, b \times d)$
  endif
endif
if isnan(l) or isnan(u)
  if not (isnan(a) or isnan(c)) {Both are not empty }
    $l = -\infty$
    $u = +\infty$
  endif
endif
Restore_Rounding_Mode ;
return $[l, u]$

Interval division

Input: X= [a,b], Y = [c,d]
{ Intervals $[-\infty, -\infty]$ and $[+\infty, +\infty]$ cannot occur by construction }
Output: X / Y
if (isnan(a) or isnan(c)) {One or both is empty }
  return $[NaN_{\emptyset}, NaN_{\emptyset}]$
else if $c \leq 0$ and $d \geq 0$ { $0 \in Y$ }
  return $[-\infty, +\infty]$
endif
Save_Rounding_Mode ;
Set_Rounding_Mode_Up;
if $a \leq 0$ and $b \geq 0$ { $0 \in X$}
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a/c)$
    $u = b/c$
  else { **if** $d < 0$ } $\{Y < 0\}$
    $l = -(-b/d)$
    $u = a/d$
  endif
else if $a > 0$ $\{X > 0\}$
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a/d)$
    $u = b/c$
  else { **if** $d < 0$ } $\{Y < 0\}$
    $l = -(-b/d)$
    $u = a/c$
  endif
else { **if** $b < 0$ } $\{X < 0\}$
  if $c > 0$ $\{Y > 0\}$
    $l = -(-a/c)$
    $u = b/d$
  else { $d < 0$ } $\{Y < 0\}$
    $l = -(-b/c)$
    $u = a/d$
  endif
endif
Restore_Rounding_Mode ;
return $[l, u]$

References

[1] Abramowitz, M., and Stegun. I., Handbook of Mathematical Functions, Dover Publications, Inc., New York, 1972.

[2] Corbett, R.P., *Enhanced Arithmetic for Fortran*, SIGPLAN Notices, Vol. 17, No. 12, 1982

[3] ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, 1985.

[4] Hansen, E.R, *Global Optimization Using Interval Analysis,* Marcel Dekker, Inc., New York, N.Y., 1992.

[5] Hough, D., *post to numeric-interest@validgh.com mailing list*, 1995.

[6] ISO/IEC JTC1/SC22/WG5 - N1231, Technical Report for Floating Point Exception Handling.

[7] ANSI X3J3 1996-156, Interval Arithmetic—The Data Type and Low-Level Operations, 1996.

[8] X3J3/97-155, ISO/IEC JTC1/SC22/WG5 - N1231, Proposed Syntax– Exceptions for Interval Intrinsic Functions.

[9] Kearfott, R. B., et al. A specific proposal for interval arithmetic in Fortran, 1996. http://interval.usl.edu/F90/f96-pro.asc

[10] Krämer, W., Kulisch, U., Lohner, R., *Numerical Toolbox for Verified computing II. Advanced Numerical Problems*, Springer–Verlag, 1998. (to appear)

[11] M90 Reference Manual. Minnesota Fortran 1990 Standards Version Edition 1. University of Minnesota, 1983.

[12] Moore, R.E., Interval Analysi, Prentice-Hall, Englewood Cliffs, N.J., 1966.

[13] Moore, R.E., *"Methods and applications of interval analysis"*, SIAM Studies in Applied Mathematics. SIAM, Philadelphia, Pennsylvania, 1979.

[14] Popova, E., *Interval Operations Involving NaNs*, Reliable Computing Vol. 2, No. 2, pp. 161–165.

[15] Sun Workshop, Numerical Computations Guide, 1996.

[16] Sendov, B., *Some topics of segment analysis*, In: Interval Mathematics, Ed. Nickel, K., Academic Press, pp. 203–222, 1980.

[17] Schulte, M., Zelov, V., Walster, G.W., and Chiriaev, D., *Single Number Interval I/O*, Proceedings of SCAN–98 Conference, 1999.

[18] Walster, G.W., Hansen, E.R, and Pryce, J.D., *Extended Real Intervals and the Topological Closure of Extended Real Numbers*, Manuscript, 1999.

[19] Walster, G.W., *Interval Angles*, Manuscript, 1998.

[20] Walster, G.W., *The "Simple" Closed Interval System*, Manuscript, 1999.

[21] Walster, G.W. *Compiler Support for Interval Arithmetic with Inline Code Generation and Nonstop Exception Handling*, Manuscript, 1998.

[22] Walster, G.W., *Philosophy and Practicalities of Interval Arithmetic*, in Moore (Ed.) Reliability in Computing. Academic Press, Inc., San Diego, California, pp. 309–323, 1988.

[23] Walster, G.W., and Hansen, E.R., *Interval Algebra, Composite Functions and Dependence in Compilers*, Manuscript, 1997.
http://www.mscs.mu.edu/ georgec/Classes/GlobSol/Papers/composite.ps

[24] Yakovlev, A.G., *Classification approach to programming of localizational (interval) computations*, Interval Computations, **1**(3), 1992.