Interval Arithmetic Programming Reference

Sun WorkShop 6 Fortran 95

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-3595-10
May 2000, Revision A

Send comments about this document to: `docfeedback@sun.com`

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Tables

Code Samples

Preface

This manual documents the intrinsic interval data types in the Sun™ WorkShop 6 Fortran 95 compiler (`f95`).

Who Should Use This Book

This is a *reference* manual intended for programmers with a working knowledge of the Fortran language, the Solaris™ operating environment, and UNIX commands.

What Is in This Book

This book contains the following two chapters:

Chapter 1, "Using Interval Arithmetic With `f95`," describes the goals for intrinsic interval support in `f95` and provides code samples that interval programmers can use to learn more about the interval features in `f95`.

Chapter 2, "`f95` Interval Reference," describes the interval language extensions to `f95`.

"Glossary," contains definitions of interval terms.

What Is Not in This Book

This book is not an introduction to intervals and does not contain derivations of the interval innovations included in `f95`. For a list of sources containing introductory interval information, see the Interval Arithmetic README.

What Typographic Changes Mean

The following table describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example

`AaBbCc123`	Code samples, the names of commands, files, and directories; on-screen computer output	`INTERVAL(4):: X = [2,3]`
		`PRINT *, "X = ", X`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`my_system%` **`f95 -xia test.f95`**
		`my_system%` **`a.out`**
		`X = [2.0,3.0]`
		`my_system%`

`AaBbCc123`	Placeholders for `INTERVAL` language elements	The `INTERVAL` affirmative order relational operators *`op`* ∈ {`LT`, `LE`, `EQ`, `GE`, `GT`} are equivalent to the mathematical operators $\mathrm{op} \in \{ <, \leq, =, \geq, > \}$.
AaBbCc123	Variables used in equations, book titles, new words or terms, or words to be emphasized	The Fortran code equivalent of $X \cap Y$ is `X .IX. Y`
AaBbCc123	Command-line placeholder: replace with a real name or value	To invoke `f95` with intrinsic `INTERVAL` support, type `my_system%` **`f95 -xia`** *source_file***`.f95`**

Note – Examples use `math%` as the system prompt.

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	`%`
Bourne shell and Korn shell	`$`
C shell, Bourne shell, and Korn shell superuser	`#`

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your `PATH` environment variable:

1. **Display the current value of the `PATH` variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing `/opt/SUNWspro/bin/`.**

   If you find the paths, your `PATH` variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your `PATH` environment variable by following the instructions in this section.

To determine if you need to set your MANPATH environment variable:

1. **Request the workshop man page by typing:**

```
% man workshop
```

2. **Review the output, if any.**

   If the workshop(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in this section for setting your MANPATH environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the /opt directory. Contact your system administrator if your Sun WorkShop software is not installed in /opt.

The PATH and MANPATH variables should be set in your home .cshrc file if you are using the C shell or in your home .profile file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your PATH variable:

  `/opt/SUNWspro/bin`

- To access Sun WorkShop man pages with the man command, add the following to your MANPATH variable:

  `/opt/SUNWspro/man`

For more information about the PATH variable, see the csh(1), sh(1), and ksh(1) man pages. For more information about the MANPATH variable, see the man(1) man page. For more information about setting your PATH and MANPATH variables to access this release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

Related Interval References

The interval literature is large and growing. Interval applications exist in various substantive fields. However, most interval books and journal articles either contain new interval algorithms, or are written for interval analysts who are developing new interval algorithms. There is not yet a book titled "Introduction to Intervals."

The Sun WorkShop 6 `f95` compiler is not the only source of support for intervals. Readers interested in other well known sources can refer to the following books:

- IBM High Accuracy Arithmetic - Extended Scientific Computation (ACRITH-XSC), General Information, GC 33-6461-01 IBM Corp., 1990.
- R.Klatte, U.Kulisch, M.Neaga, D.Ratz, Ch.Ullrich, PASCAL-XSC Language Reference with Examples, Springer, 1991.
- R.Klatte, U.Kulisch, A.Wiethoff, C.Lawo, M. Rauch, C-XSC Class library for Extended Scientific Computing, Springer, 1993.
- R.Hammer, M.Hocks, U.Kulisch, D.Ratz, Numerical Toolbox for Verified Computing I, Basic Numerical Problems, Springer, 1993.

For a list of technical reports that establish the foundation for the interval innovations implemented in `f95`, see "References" on page 126. See the Interval Arithmetic README for the location of the online versions of these references.

Online Resources

Additional interval information is available at various web sites and email mailing lists. For a list of online resources, refer to the Interval Arithmetic README.

Web Sites

A detailed bibliography and interval FAQ can be obtained online at the URLs listed in the Interval Arithmetic README.

Email

To discuss interval arithmetic issues or ask questions regarding the use of interval arithmetic, a mailing list has been constructed. Anyone can send questions to this list. Refer to the Interval Arithmetic README for instructions on how to subscribe to this mailing list.

To report a suspected interval error, send email to

```
sun-dp-comments@Sun.COM
```

Include the following text in the Subject line or the email message:

`WORKSHOP "6.0` *mm/dd/yy*`" Interval`

where *mm/dd/yy* is the month, day, and year.

Code Examples

All code examples in this book are contained in the following directory:

```
/opt/SUNWspro/examples/intervalmath/docExamples
```

The name of each file is `ce`*n-m*`.f95` where *n* is the chapter in which the example occurs and *m* is the number of the example. Additional interval examples can be found in the following directory:

```
/opt/SUNWspro/examples/intervalmath/general
```

Related Non-Interval Sun WorkShop 6 Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the `docs.sun.com`sm Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  ```
  http://docs.sun.com
  ```

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:
  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

    `/opt/SUNWspro/docs/index.html`

  (Contact your system administrator if your Sun WorkShop software is not installed in the `/opt` directory.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte Developer 6 / Sun WorkShop 6	*Analyzing Program Performance With Sun WorkShop 6*	Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool `er_print`, the LoopTool and LoopReport utilities, and UNIX profiling tools `prof`, `gprof`, and `tcov`.
	Debugging a Program With dbx	Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.
	Introduction to Sun WorkShop	Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.
Forte™ C 6 / Sun WorkShop 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.
Forte™ C++ 6 / Sun WorkShop 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.
Forte™ TeamWare 6 / Sun WorkShop TeamWare 6	*Sun WorkShop TeamWare 6 User's Guide*	Describes how to use the Sun WorkShop TeamWare code management tools.
Forte Developer 6 / Sun WorkShop Visual 6	*Sun WorkShop Visual User's Guide*	Describes how to use Visual to create C++ and Java™ graphical user interfaces.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	**Document Title**	**Description**
Forte™ / Sun Performance Library 6	*Sun Performance Library Reference*	Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.
	Sun Performance Library User's Guide	Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.
Numerical Computation Guide	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ Class Library Reference*	Provides details on the `Tools.h++` class library.
	Tools.h++ User's Guide	Discusses use of the C++ classes for enhancing the efficiency of your programs.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	**Document Title**	**Description**
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

CHAPTER **1**

Using Interval Arithmetic With `f95`

1.1 `f95` `INTERVAL` Type and Interval Arithmetic Support

Interval arithmetic is a system for computing with intervals of numbers. Because interval arithmetic always produces intervals that contain the set of all possible result values, interval algorithms have been developed to perform surprisingly difficult computations. For more information on interval applications, see the Interval Arithmetic README.

Since the inception of interval arithmetic, interval algorithms that produce narrow-width results have been developed, and the syntax and semantics for interval language support have been designed. However, relatively little progress has been made in providing commercially available and supported interval compilers. With one exception (M77 Minnesota FORTRAN 1977 Standards Version Edition 1), interval systems have been based on pre-processors, C++ classes, or Fortran 90 modules. The goals of intrinsic compiler support for interval data types in `f95` are:

- Reliability
- Speed
- Ease-of-use

Interval support in the Sun WorkShop 6 release of `f95` is a significant extension to Fortran.

1.2 f95 Interval Support Goal: Implementation Quality

The goal of intrinsic INTERVAL support in f95 is to stimulate development of commercial interval solver libraries and applications by providing program developers with:

- Quality interval code
- Narrow-width interval results
- Rapidly executing interval code
- An easy to use interval software development environment that includes interval-specific language support and compiler features

Support and features are components of implementation quality. Throughout this book, various quality of implementation opportunities are described. Additional suggestions from users are welcome.

1.2.1 Quality Interval Code

As a consequence of evaluating any interval expression, a valid interval-supporting compiler must produce an interval that contains the set of all possible results. The requirement to contain the set of all possible results is called the containment constraint of interval arithmetic. The failure to satisfy the containment constraint is a containment failure. A silent failure (with no warning or documentation) to satisfy the interval containment constraint is a fatal error in any interval computing system. By satisfying this single constraint, intervals provide unprecedented computing quality.

Given the containment constraint is satisfied, implementation quality is determined by the location of a point in the two-dimensional plane whose axes are *runtime* and *interval width*. On both axes, small is better. How to trade runtime for interval width depends on the application. Both runtime and interval width are obvious measures of interval-system quality. Because interval width and runtime are always available, measuring the accuracy of both interval algorithms and implementation systems is no more difficult than measuring their speed.

The Sun WorkShop 6 tools for performance profiling can be used to tune interval programs. However, in f95, no interval-specific tools exist to help isolate where an algorithm may gain unnecessary interval width. As described in Section 1.4 “Code Development Tools” on page 35, some interval dbx and global program checking (GPC) support are provided. Adding additional interval-specific code development and debugging tools are quality of implementation opportunities.

1.2.2 Narrow-Width Interval Results

All the normal language and compiler quality of implementation opportunities exist for intervals, including rapid execution and ease-of-use.

Valid interval implementation systems include a new additional quality of implementation opportunity: Minimize the width of computed intervals while always satisfying the containment constraint.

If an interval's width is as narrow as possible, it is said to be *sharp*. For a given floating-point precision, an interval result is sharp if its width is as narrow as possible.

The following can be said about the width of intervals produced by the `f95` compiler:

- Individual intervals are sharp approximations of constants.
- Individual interval arithmetic operators produce sharp results.
- Intrinsic mathematical functions usually produce sharp results.

1.2.3 Rapidly Executing Interval Code

By providing compiler optimization and hardware instruction support, `INTERVAL` operations are not necessarily slower than their `REAL` floating-point counterparts. In `f95`, the following can be said about the speed of intrinsic interval operators and mathematical functions:

- Arithmetic operations are reasonably fast.
- The speed of default `INTERVAL` mathematical functions is generally less than 2 times that of their `DOUBLE PRECISION` counterparts. `KIND = 4` intrinsic interval math functions are provided, but are not tuned for speed (unlike their `KIND = 8` counterparts). `KIND = 16` mathematical functions are not provided in this release. However, other `INTERVAL KIND = 16` functions are supported.
- The following intrinsic `INTERVAL` array functions are optimized for performance:
  - `SUM`
  - `PRODUCT`
  - `DOT_PRODUCT`
  - `MATMUL`

1.2.4 Easy to Use Development Environment

The intrinsic `INTERVAL` data type in Fortran facilitates interval code development, testing, and execution. To make interval code transparent (easy to write and read), interval syntax and semantics have been added to Fortran. User acceptance will ultimately determine which interval features are added to standard Fortran.

By introducing intervals as an intrinsic data type to Fortran, all of the applicable syntax and semantics of Fortran become immediately available. Sun WorkShop 6 `f95` includes the following interval Fortran extensions:

- `INTERVAL` data types
- `INTERVAL` arithmetic operations and intrinsic mathematical functions form a closed mathematical system. (This means that valid results are produced for any possible operator-operand combination, including division by zero and other indeterminate forms involving zero and infinities.)
- Three classes of interval relational operators:
  - Certainly
  - Possibly
  - Set
- Intrinsic `INTERVAL`-specific operators, such as `.IX.` (intersection) and `.IH.` (interval hull)
- `INTERVAL`-specific functions, such as `INF`, `SUP`, and `WID`
- `INTERVAL` input/output, including single-number input/output
- Expression-context-dependent `INTERVAL` constants
- Mixed-mode interval expression processing

For examples and more information on these and other intrinsic interval functions, see CODE EXAMPLE 1-11 through CODE EXAMPLE 1-14 and Section 2.9.4.4 “Intrinsic Functions” on page 120.

Chapter 2 describes these and other interval features.

1.3 Writing Interval Code for `f95`

The examples in this section are designed to help new interval programmers to understand the basics and to quickly begin writing useful interval code. Modifying and experimenting with the examples is strongly recommended.

All code examples in this book are contained in the directory:

```
/opt/SUNWspro/examples/intervalmath/docExamples
```

The name of each file is `ce`*n*-*m*`.f95`, where *n* is the chapter in which the example occurs, and *m* is the number of the example. Additional interval examples are contained in the directory:

```
/opt/SUNWspro/examples/intervalmath/general
```

1.3.1 Command-Line Options

Including the following command-line macro in the `f95` command line invokes recognition of `INTERVAL` data types as intrinsic and controls `INTERVAL` expression processing:

- Compiler support for widest-need interval expression processing is invoked by including:
  `-xia` or `-xia=widestneed`
- Compiler support for strict interval expression processing is invoked by including:
  `-xia=strict`

For intrinsic `INTERVAL` data types to be recognized by the compiler, either `-xia` or `-xinterval` must be entered in the `f95` command line.

All command-line options that interact with intervals are described in Section 2.3.3 "Interval Command-Line Options" on page 52. Widest-need and strict expression processing are described in Section 2.3 "`INTERVAL` Arithmetic Expressions" on page 48.

The simplest command-line invocation of `f95` with interval support is shown in CODE EXAMPLE 1-1.

1.3.2 Hello Interval World

Unless explicitly stated otherwise, all code examples are compiled using the `-xia` command-line option. The `-xia` command-line option is required to use the interval extensions to `f95`.

CODE EXAMPLE 1-1 is the interval equivalent of "hello world."

CODE EXAMPLE 1-1 Hello Interval World

```
math% cat ce1-1.f95
PRINT *, "[2, 3] + [4, 5] = ", [2, 3] + [4, 5]    ! line 1
END

math% f95 -xia ce1-1.f95
math% a.out
 [2, 3] + [4, 5] =  [6.0,8.0]
```

CODE EXAMPLE 1-1 uses list-directed output to print the labeled sum of the intervals [2, 3] and [4, 5].

1.3.3 Interval Declaration and Initialization

The `INTERVAL` declaration statement performs the same functions for `INTERVAL` data items as the `REAL`, `INTEGER`, and `COMPLEX` declarations do for their respective data items. The default `INTERVAL` kind type parameter value (KTPV) is twice the default `INTEGER` KTPV. This permits any default `INTEGER` to be exactly represented using a degenerate default `INTERVAL`. See Section 1.3.7 "Default Kind Type Parameter Value (KTPV)" on page 23 for more information.

CODE EXAMPLE 1-2 uses `INTERVAL` variables and initialization to perform the same operation as CODE EXAMPLE 1-1.

CODE EXAMPLE 1-2 Hello Interval World with `INTERVAL` Variables

```
math% cat ce1-2.f95
INTERVAL :: X = [2, 3], Y = [4, 5]   ! Line 1
PRINT *, "[2, 3] + [4, 5] = ", X+Y   ! Line 2
END
math% f95 -xia ce1-2.f95
math% a.out
 [2, 3] + [4, 5] =  [6.0,8.0]
```

In line 1, the variables, `X` and `Y` are declared to be default type `INTERVAL` variables and are initialized to [2, 3] and [4, 5], respectively. Line 2 uses list-directed output to print the labeled interval sum of `X` and `Y`.

1.3.4 `INTERVAL` Input/Output

Full support for reading and writing intervals is provided. Reading and writing `INTERVAL` and `COMPLEX` data items are similar. Intervals use square brackets, instead of parentheses as delimiters. Because reading and interactively entering interval data can be tedious, a *single-number* interval format is introduced. The single-number convention is that any number not contained in brackets is interpreted as an interval whose lower and upper bounds are constructed by subtracting and adding 1 unit to the last displayed digit.

Thus

$2.345 = [2.344, 2.346]$,

$2.34500 = [2.34499, 2.34501]$,

and

$23 = [22, 24]$.

Symbolically,

$[2.34499, 2.34501] = 2.34500 + [-1, +1]_{\text{uld}}$

where $[-1, +1]_{\text{uld}}$ means that the interval [-1, +1] is added to the last digit of the preceding number. The subscript, uld, is a mnemonic for "unit in the last digit."

To represent a degenerate interval, a single number can be enclosed in square brackets. For example,

$[2.345] = [2.345, 2.345] = 2.345000000000.....$

This convention is used both for input and representing degenerate literal `INTERVAL` constants in Fortran code. Thus, type **`[0.1]`** to indicate the input value is an exact decimal number, even though 0.1 is not machine representable.

For example, during input to a program, `[0.1,0.1] = [0.1]` represents the *point*, 0.1, while using single-number input/output, `0.1` represents the interval

$0.1 + [-1, +1]_{\text{uld}} = [0, 0.2]$.

In `f95` the input conversion process constructs a sharp interval that contains the input decimal value. If the value is machine representable, the internal machine approximation is degenerate. If the value is not machine representable, an interval having width of 1-ulp (unit-in-the-last-place of the mantissa) is constructed.

Note – A uld and an ulp are different. A uld refers to implicitly constructing an interval using the single number input/output format to add and subtract one unit to and from the last displayed digit. An ulp is the smallest possible increment or decrement that can be made to an internal machine number.

The simplest way to read and print `INTERVAL` data items is with list-directed input and output.

CODE EXAMPLE 1-3 is a simple tool to help users become familiar with interval arithmetic and single-number `INTERVAL` input/output using list-directed `READ` and `PRINT` statements. Complete support for formatted `INTERVAL` input/output is provided, as described in Section 2.9.2 "Input and Output" on page 98.

Note – The interval containment constraint requires that directed rounding be used both during input and output. With single-number input followed immediately by single-number output, a decimal digit of accuracy can appear to be lost. In fact, the width of the input interval is increased by at most 1-ulp, when the input value is not machine representable. See Section 1.3.5 "Single-Number Input/Output" on page 19 and CODE EXAMPLE 1-6 on page 22.

CODE EXAMPLE 1-3 Interval Input/Output

```
math% cat ce1-3.f95
   INTERVAL ::  X, Y
   INTEGER  :: IOS = 0
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE = 'NO')
   READ(*, *, IOSTAT = IOS) X, Y
   DO WHILE (IOS >= 0)
       PRINT *, " For X =", X, ", and Y =", Y
       PRINT *, "X+Y =", X+Y
       PRINT *, "X-Y =", X-Y
       PRINT *, "X*Y =", X*Y
       PRINT *, "X/Y =", X/Y
       PRINT *, "X**Y =", X**Y
       WRITE(*, 1, ADVANCE = 'NO')
       READ(*, *, IOSTAT=IOS) X, Y
   END DO
1  FORMAT(" X, Y = ? ")
   END
```

CODE EXAMPLE 1-3 Interval Input/Output *(Continued)*

```
%math f95 -xia ce1-3.f95
%math a.out
 Press Control/D to terminate!
 X, Y = ? [1,2] [3,4]
 For X = [1.0,2.0] , and Y = [3.0,4.0]
 X+Y = [4.0,6.0]
 X-Y = [-3.0,-1.0]
 X*Y = [3.0,8.0]
 X/Y = [0.25,0.66666666666666675]
 X**Y = [1.0,16.0]
 X, Y = ? [1,2] -inf
 For X = [1.0,2.0] , and Y = [-Inf,-1.7976931348623157E+308]
 X+Y = [-Inf,-1.7976931348623155E+308]
 X-Y = [1.7976931348623157E+308,Inf]
 X*Y = [-Inf,-1.7976931348623157E+308]
 X/Y = [-1.1125369292536012E-308,0.0E+0]
 X**Y = [0.0E+0,Inf]
 X, Y = ? <Control-D>
```

1.3.5 Single-Number Input/Output

One of the most frustrating aspects of reading interval output is comparing interval infima and suprema to count the number of digits that agree. For example, CODE EXAMPLE 1-4 and CODE EXAMPLE 1-5 shows the interval output of a program that generates different random width `INTERVAL` data.

Note – Only program output is shown in CODE EXAMPLE 1-4 and CODE EXAMPLE 1-5. The code that generates the output is included with the examples located in the `/opt/SUNWspro/examples/intervalmath/docExamples` directory.

CODE EXAMPLE 1-4 [inf, sup] Interval Output

```
%math f95 -xia ce1-4.f95
%math a.out
Press Control/D to terminate!
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,4,0
[ 0.2017321E-029, 0.2017343E-029]
[ 0.2176913E-022, 0.2179092E-022]
[-0.3602303E-006,-0.3602302E-006]
[-0.3816341E+038,-0.3816302E+038]
[-0.1011276E-039,-0.1011261E-039]
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,8,0
[ -0.3945547546440221E+035, -0.3945543600894656E+035]
[  0.5054960140922359E-270,  0.5054960140927415E-270]
[ -0.2461623589326215E-043, -0.2461623343163864E-043]
[ -0.2128913523672577E+204, -0.2128913523672576E+204]
[ -0.3765492464030608E-072, -0.3765492464030606E-072]
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,16,0
[  0.199050353252318620256245071374058E+055,
0.199050353252320610759742664557447E+055]
[ -0.277386431989417915223682516437493E+203,
-0.277386431989417915195943874118822E+203]
[  0.132585288598265472316856821380503E+410,
0.132585288598265472316856822706356E+410]
[  0.955714436647437881071727891682804E+351,
0.955714436647437881071727891683760E+351]
[ -0.224211897768824210398306994401732E+196,
-0.224211897768824210398306994177519E+196]
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: <Control-D>
```

Compare the output readability in CODE EXAMPLE 1-4 with CODE EXAMPLE 1-5.

CODE EXAMPLE 1-5 Single-Number Output.

```
%math a.out
 Press Control/D to terminate!
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,4,1
     0.20173  E-029
     0.218    E-022
    -0.3602303E-006
    -0.38163  E+038
    -0.10112  E-039
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,8,1
     -0.394554          E+035
      0.505496014092    E-270
     -0.2461623         E-043
     -0.2128913523672577E+204
     -0.3765492464030607E-072
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: 5,16,1
         0.19905035325232                  E+055
        -0.27738643198941791 52             E+203
         0.1325852885982654723168568 22      E+410
         0.95571443664743788107172789 1683   E+351
        -0.22421189776882421039830699 4      E+196
Enter number of intervals, KTPV (4,8,16) and 1 for single-number
output: <Control-D>
```

In the single-number display format, trailing zeros are significant. See Section 2.9.2 “Input and Output” on page 98 for more information.

Intervals can always be entered and displayed using the traditional [*inf*, *sup*] display format. In addition, a single number in square brackets denotes a point. For example, on input, `[0.1]` is interpreted as the number 1/10. To guarantee containment, directed rounding is used to construct an internal approximation that is known to contain the number 1/10.

CODE EXAMPLE 1-6 Character Input with Internal Data Conversion

```
math% cat ce1-6.f95
INTERVAL :: X
   INTEGER  :: IOS = 0
   CHARACTER*30 BUFFER
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, '(A12)', IOSTAT=IOS) BUFFER
   DO WHILE (IOS >= 0)
     PRINT *, ' Your input was: ', BUFFER
     READ(BUFFER, '(Y12.16)') X
     PRINT *, "Resulting stored interval is:", X
      PRINT '(A, Y12.2)', ' Single number interval output  is:', X
     WRITE(*, 1, ADVANCE='NO')
     READ(*, '(A12)', IOSTAT=IOS) BUFFER
   END DO
1  FORMAT(" X = ? ")
   END
math% f95 -xia ce1-6.f95
math% a.out
 Press Control/D to terminate!
 X = ? 1.37
 Your input was: 1.37
 Resulting stored interval is:
[1.3599999999999998,1.3800000000000002]
 Single number interval output  is:  1.3
 X = ? 1.444
 Your input was: 1.444
 Resulting stored interval is:
[1.4429999999999998,1.4450000000000001]
 Single number interval output  is:  1.44
 X = ? <Control-D>
```

CODE EXAMPLE 1-6 notes:

- Single numbers in square brackets represent degenerate intervals.
- When a non-machine representable number is read using single-number input, conversion from decimal to binary (radix conversion) and the containment constraint force the number’s interval width to be increased by 1-ulp (unit in the last place of the mantissa). When this result is displayed using single-number

output, it can appear that a decimal digit of accuracy has been lost. This is not so. To echo single-number interval inputs, use character input together with internal READ statement data conversion, as shown in CODE EXAMPLE 1-6 on page 22.

Note – The empty interval is supported in f95. The empty interval can be entered as "[empty]". Infinite interval endpoints are also supported, as described in Section 2.9.2.1 "External Representations" on page 98 and illustrated in CODE EXAMPLE 2-37 on page 100.

1.3.6 Interval Statements and Expressions

The f95 compiler contains the following INTERVAL-specific statements, expressions, and extensions:

- The INTERVAL data type, related instructions, and statements described in TABLE 1-1 on page 24 are supported.
- All intrinsic functions that accept real arguments have corresponding interval versions. (It is a known error that the same is not true for integers. See Section 1.5.1.1 "Integer Overflow" on page 40.)
- A number of intrinsic INTERVAL-specific functions and operators have been added, including INTERVAL-specific relational operators and set-theoretic functions. For a complete list of intrinsic INTERVAL functions and INTERVAL operators, see Section 2.9.3 "Intrinsic INTERVAL Functions" on page 116 and Section 2.9.4 "Mathematical Functions" on page 116.

1.3.7 Default Kind Type Parameter Value (KTPV)

In f95 the default INTEGER KTPV is KIND(0) = 4. To represent any default INTEGER with a degenerate default INTERVAL requires the default INTERVAL KTPV, KIND([0]), to be 2*KIND(0) = 8. Choosing 8 for the default INTERVAL KTPV is also done because:

- Intervals are often used to perform numerically intense computations, as have been performed on CDC and Cray machines.
- When evaluating a single arithmetic expression, the width of intervals necessarily grows because of accumulated rounding errors, dependence, and cancellation. Extra precision can help to reduce the effect of accumulated rounding errors.

TABLE 1-1 INTERVAL Specific Statements and Expressions

Statement/expression	Description

INTERVAL
INTERVAL(4)
INTERVAL(8)
INTERVAL(16)
	Default INTERVAL type declaration
	KIND=4 INTERVAL
	KIND=8 INTERVAL
	KIND=16 INTERVAL

[*a*,*b*] *See Note 1*	Literal INTERVAL constant: [*a*,*b*]
[*a*] *See Note 2*	[*a*,*a*]
INTERVAL A PARAMETER A=[*c*,*d*]	Named constant: A
V = *expr* *See Note 3*	Value assignment
FORMAT(E, EN, ES, F, G, VE, VF, VG, VEN, VES, Y) *See Note 4*	E, EN, ES, F, G, VE, VF, VG, VEN, VES, Y edit descriptors

(1) The letters *a* and *b* are placeholders for literal decimal constants, such as 0.1 and 0.2.

(2) A single decimal constant contained in square brackets denotes a degenerate INTERVAL constant. The same convention is used in input/output.

(3) Let *expr* stand for any Fortran arithmetic expression, whether or not it contains items of type INTERVAL. An assignment statement, V = *expr*, evaluates the expression, *expr*, and assigns the resulting value to V. Mixed-mode INTERVAL expressions are not permitted under the -xia=strict command line option. Under the -xia or -xia=widestneed option, mixed-mode expressions are correctly evaluated using widest-need expression processing. Before expression evaluation under widest-need, all integer and floating-point data items are promoted to containing intervals with the largest KTPV found anywhere in the expression, including, V. For details, see Section 2.3.2 “Value Assignment” on page 49.

(4) Interval input/output support is designed to provide flexibility, readability, and ease of code development. The most important new edit descriptor is Y, which is used to read and display intervals using the single-number interval format. For a complete description of all edit descriptors that can process intervals, see Section 2.9.2 “Input and Output” on page 98.

1.3.8 Value Assignment V = *expr*

The INTERVAL assignment statement assigns the value of an interval expression, denoted by the placeholder *expr*, to an INTERVAL variable, array element, or array, V. The syntax is:

```
V = expr
```

where V must have an INTERVAL type, and *expr* denotes any non-COMPLEX numeric expression. Under widest-need expression processing, the expression *expr* need not be an INTERVAL expression. Under strict expression processing, *expr* must be an INTERVAL expression with the same KTPV as V.

1.3.9 Mixed-Type Expression Evaluation

Gracefully handling mixed-type INTERVAL expressions is an important ease-of-use feature, because it facilitates writing transparent (easy to understand) mathematical expressions.

Mixed-type INTERVAL expressions are supported to make writing and reading interval code no more difficult than it is for REAL code. The interval containment constraint is satisfied in mixed-mode expressions using either *widest-need* or *strict* expression processing.

1.3.9.1 Widest-Need and Strict Expression Processing

Computing narrow-width interval results is facilitated if the width of INTERVAL constants is dynamically defined by expression context, as described in Section 2.3 "INTERVAL Arithmetic Expressions" on page 48. In mixed-KTPV expressions, shown in CODE EXAMPLE 1-7, dynamically increasing the KTPV of INTERVAL variables can also decrease the width of INTERVAL expression results.

CODE EXAMPLE 1-7 Mixed Precision with Widest-Need

```
math% cat ce1-7.f95
INTERVAL(4) :: X = [1, 2], Y = [3, 4]
INTERVAL    :: Z1, Z2

! Widest-need Code
Z1 = X*Y                                        ! Line 3

! Equivalent Strict Code
Z2 = INTERVAL(X, KIND=8)*INTERVAL(Y, KIND=8)    ! Line 4
IF (Z1 .SEQ. Z2)  PRINT *, 'Check.'
END
math% f95 -xia ce1-7.f95
math% a.out
 Check.
```

In line 3, $KTPV_{max}$ = KIND(Z) = 8. This value is used to promote the KTPV of X and Y to 8 before computing their product and storing the result in Z1.

These steps are shown explicitly in the equivalent strict code in line 4.

The process of scanning a statement to determine the maximum KTPV and performing the necessary promotions, is called widest-need expression processing, see Section 2.3 "INTERVAL Arithmetic Expressions" on page 48.

For syntax and semantics of the intrinsic INTERVAL constructor functions, see Section 2.8 “Extending Intrinsic INTERVAL Operators” on page 70.

1.3.9.2 Mixed-Mode (Type and KTPV) Expressions

If the widest-need principle is used with both KTPVs and data types, mixed-mode (type and KTPV) INTERVAL expressions can be safely and predictably evaluated. For example, in CODE EXAMPLE 1-8 on page 26, the expression for Y1 in line 3 is an interval expression, because X and Y1 are INTERVAL variables.

CODE EXAMPLE 1-8 Mixed Types with Widest-Need

```
math% cat ce1-8.f95
INTERVAL(16) :: X = [0.1, 0.3]
INTERVAL(4)  :: Y1, Y2

! Widest-need code
 Y1 = X + 0.1                               ! Line 3

! Equivalent strict code
 Y2 = INTERVAL(X + [0.1_16], KIND=4)        ! Line 4
 IF (Y1 == Y2) PRINT *, "Check"
END

math% f95 -xia ce1-8.f95
math% a.out
 Check
```

To guarantee containment, a containing interval must be used in place of a real approximation to the constant 0.1. However, $\mathrm{KTPV}_{\mathrm{max}}$ = 16, because KIND(X) = 16. Therefore, the INTERVAL constant [0.1_16], a sharp KTPV = 16 interval containing the exact value, 1/10, is used to update X. Finally, the result is converted to a KTPV = 4 containing interval and assigned to Y1. Line 4 contains the equivalent strict code. Under strict expression processing, neither mixed-type nor mixed-KTPV expressions are permitted.

The logical steps in widest-need expression processing are:

1. **Scan the entire statement, including the left-hand side, for any INTERVAL data items.**

   The presence of any INTERVAL constants, variables, or intrinsic functions, makes the expression’s type INTERVAL.

2. **Scan the INTERVAL expressions for $\mathrm{KTPV}_{\mathrm{max}}$, based on the KTPV of each INTERVAL, REAL, INTEGER, constant, or variable.**

Note – Integers are converted to intervals with twice their KTPV so all integer values can be exactly represented.

---

3. **Promote all variables and constants to intervals with KTPV_{max}.**
4. **Evaluate the expression.**
5. **Convert the result to a lower KTPV if needed to match the left-hand side's KTPV.**
6. **Assign the resulting value to the left-hand side.**

These steps guarantee that mixed-mode `INTERVAL` expression processing satisfies the containment constraint and efficiently produces reasonably narrow interval results.

Mixed-mode `INTERVAL` expression evaluation using widest-need expression processing is supported by default with the `-xia` command-line flag. Using `-xia=strict` eliminates any automatic type conversions to intervals and any automatic KTPV increases of `INTERVAL` variables. In strict mode, all interval type and precision conversions must be explicitly coded.

1.3.10 Arithmetic Expressions

Writing arithmetic expressions that contain `INTERVAL` data items is simple and straightforward. Except for `INTERVAL` literal constants and intrinsic `INTERVAL`-specific functions, `INTERVAL` expressions look like `REAL` arithmetic expressions. In particular, with widest-need expression processing, `REAL` and `INTEGER` variables and literal constants can be freely used anywhere in an `INTERVAL` expression, such as in CODE EXAMPLE 1-9.

CODE EXAMPLE 1-9 Simple `INTERVAL` Expression Example

```
math% cat ce1-9.f95
INTEGER  :: N = 3
REAL     :: A = 5.0
INTERVAL :: X

X = 0.1*A/N                     ! Line 5
PRINT *, "0.1*A/N = ", X
END

math% f95 -xia ce1-9.f95
math% a.out
 0.1*A/N =  [0.16666666666666662,0.16666666666666672]
```

Because X, the variable to which the assignment is made in line 5, is an INTERVAL, the following steps are taken before evaluating the expression `0.1*A/N`:

1. The literal constant 0.1 is converted to the default INTERVAL variable containing the degenerate interval `[0.1]`.

   While not required in a valid interval system implementation, Sun WorkShop 6 f95 performs sharp data conversions. For example, the internal approximation of [0.1] is 1-ulp wide.

2. The REAL variable A is converted to the degenerate interval `[5]`.

3. The INTEGER variable N is converted to the degenerate interval `[3]`.

The expression $[0.1] \times [5]/[3]$ is evaluated using interval arithmetic. The above steps are part of *widest-need* expression processing, which is required to satisfy the containment constraint when evaluating mixed-mode INTERVAL expressions. See Section 1.3.9 “Mixed-Type Expression Evaluation” on page 25.

An INTERVAL assignment statement must satisfy one requirement: the variable to which the assignment is made must be an INTERVAL variable, array element, or array. For more information on the widest-need processing mode, see Section 2.3.1 “Mixed-Mode INTERVAL Expressions” on page 48.

Because the interval system implemented in Sun WorkShop 6 f95 is closed, if any INTERVAL expression fails to produce a valid interval result, it is a compiler error that should be reported. See Section 1.4 “Code Development Tools” on page 35 for information on how to report a suspected error and Section 1.5.1 “Known Containment Failures” on page 40 for a list of known errors.

Note – Not all mathematically equivalent INTERVAL expressions produce intervals having the same width. Additionally, it is often not possible to compute a sharp result by simply evaluating a single INTERVAL expression. In general, interval result width depends on the value of INTERVAL arguments and the form of the expression.

1.3.11 Interval Order Relations

Ordering intervals is more complicated than ordering points. Testing whether 2 is less than 3 is unambiguous. With intervals, while the interval `[2,3]` is certainly less than the interval `[4,5]`, what should be said about `[2,3]` and `[3,4]`?

Three different classes of INTERVAL relational operators are implemented:

- certainly
- possibly
- set

For a certainly-relation to be *true,* every element of the operand intervals must satisfy the relation. A possibly-relation is *true* if it is satisfied by any elements of the operand intervals. The set-relations treat intervals as sets. The three classes of `INTERVAL` relational operators converge to the normal relational operators on points if both operand intervals are degenerate.

To distinguish the three operator classes, the normal two-letter Fortran relation mnemonics are prefixed with the letters `C`, `P`, or `S`. In `f95` the set operators `.SEQ.` and `.SNE.` are the only operators for which the point defaults (`.EQ.` or `==` and `.NE.` or `/=`) are supported. In all other cases, the relational operator class must be explicitly identified, as for example in:

- `.CLT.` certainly less than
- `.PLT.` possibly less than
- `.SLT.` set less than

See Section 2.4 "Intrinsic Operators" on page 55 for the syntax and semantics of all `INTERVAL` operators.

The following program demonstrates the use of a set-equality test.

CODE EXAMPLE 1-10 Set-Equality Test

```
math% cat ce1-10.f95
INTERVAL :: X = [2, 3], Y = [4, 5]        ! Line 1
IF(X+Y .SEQ. [6, 8]) PRINT *, "Check."    ! Line 2
END
math% f95 -xia ce1-10.f95
math% a.out
 Check.
```

Line 2 uses the set-equality test to verify that `X+Y` is equal to the interval [6, 8].

An equivalent line 2 is:

```
 IF(X+Y == [6, 8]) PRINT *, "Check." ! line 2
```

Use CODE EXAMPLE 1-11 on page 30 and CODE EXAMPLE 1-12 on page 31 to explore the result of `INTERVAL`-specific relational operators.

CODE EXAMPLE 1-11 Interval Relational Operators

```
math% cat ce1-11.f95
   INTERVAL ::  X, Y
   INTEGER  :: IOS = 0
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, *, IOSTAT=IOS) X, Y
   DO WHILE (IOS >= 0)
       PRINT *, " For X =", X, ", and Y =", Y
       PRINT *, 'X .CEQ. Y, X .PEQ. Y, X .SEQ. Y =', &
                 X .CEQ. Y, X .PEQ. Y, X .SEQ. Y
       PRINT *, 'X .CNE. Y, X .PNE. Y, X .SNE. Y =', &
                 X .CNE. Y, X .PNE. Y, X .SNE. Y
       PRINT *, 'X .CLE. Y, X .PLE. Y, X .SLE. Y =', &
                 X .CLE. Y, X .PLE. Y, X .SLE. Y
       PRINT *, 'X .CLT. Y, X .PLT. Y, X .SLT. Y =', &
                 X .CLT. Y, X .PLT. Y, X .SLT. Y
       PRINT *, 'X .CGE. Y, X .PGE. Y, X .SGE. Y =', &
                 X .CGE. Y, X .PGE. Y, X .SGE. Y
       PRINT *, 'X .CGT. Y, X .PGT. Y, X .SGT. Y =', &
                 X .CGT. Y, X .PGT. Y, X .SGT. Y
       WRITE(*, 1, ADVANCE='NO')
       READ(*, *, IOSTAT=IOS) X, Y
   END DO
1  FORMAT( " X, Y = ")
   END
math% f95 -xia ce1-11.f95
math% a.out
 Press Control/D to terminate!
 X, Y = [2] [3]
 For X = [2.0,2.0] , and Y = [3.0,3.0]
 X .CEQ. Y, X .PEQ. Y, X .SEQ. Y = F F F
 X .CNE. Y, X .PNE. Y, X .SNE. Y = T T T
 X .CLE. Y, X .PLE. Y, X .SLE. Y = T T T
 X .CLT. Y, X .PLT. Y, X .SLT. Y = T T T
 X .CGE. Y, X .PGE. Y, X .SGE. Y = F F F
 X .CGT. Y, X .PGT. Y, X .SGT. Y = F F F
 X, Y = 2 3
 For X = [1.0,3.0] , and Y = [2.0,4.0]
 X .CEQ. Y, X .PEQ. Y, X .SEQ. Y = F T F
 X .CNE. Y, X .PNE. Y, X .SNE. Y = F T T
 X .CLE. Y, X .PLE. Y, X .SLE. Y = F T T
 X .CLT. Y, X .PLT. Y, X .SLT. Y = F T T
 X .CGE. Y, X .PGE. Y, X .SGE. Y = F T F
 X .CGT. Y, X .PGT. Y, X .SGT. Y = F T F
 X, Y = <Control-D>
```

CODE EXAMPLE 1-12 demonstrates the use of the INTERVAL-specific operators listed in TABLE 1-2.

TABLE 1-2 Interval-Specific Operators

Operator	Name	Mathematical Symbol
.IH.	Interval Hull	$\underline{\cup}$
.IX.	Intersection	$\cap$
.DJ.	Disjoint	$A \cap B = \varnothing$
.IN.	Element	$\in$
.INT.	Interior	See Section 2.7.3 "Interior: (X .INT. Y)" on page 64.
.PSB.	Proper Subset	$\subset$
.PSP.	Proper Superset	$\supset$
.SB.	Subset	$\subseteq$
.SP.	Superset	$\supseteq$

CODE EXAMPLE 1-12 Set Operators

```
math% cat ce1-12.f95
   INTERVAL ::  X, Y
   INTEGER  :: IOS = 0
   REAL(8)  :: R = 1.5
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, *, IOSTAT=IOS) X, Y
   DO WHILE (IOS >= 0)
       PRINT *, " For X =", X, ", and Y =", Y
       PRINT *, 'X .IH.  Y =', X .IH. Y
       PRINT *, 'X .IX.  Y =', X .IX. Y
       PRINT *, 'X .DJ.  Y =', X .DJ. Y
       PRINT *, 'R .IN.  Y =', R .IN. Y
       PRINT *, 'X .INT. Y =', X .INT. Y
       PRINT *, 'X .PSB. Y =', X .PSB. Y
       PRINT *, 'X .PSP. Y =', X .PSP. Y
       PRINT *, 'X .SP.  Y =', X .SP. Y
       PRINT *, 'X .SB.  Y =', X .SB. Y

       WRITE(*, 1, ADVANCE='NO')
       READ(*, *, IOSTAT=IOS) X, Y
   END DO
1  FORMAT(" X, Y = ? ")
   END
```

CODE EXAMPLE 1-12 Set Operators *(Continued)*

```
math% f95 -xia ce1-12.f95
math% a.out
 Press Control/D to terminate!
 X, Y = ? [1] [2]
 For X = [1.0,1.0] , and Y = [2.0,2.0]
 X .IH.  Y = [1.0,2.0]
 X .IX.  Y = [EMPTY]
 X .DJ.  Y = T
 R .IN.  Y = F
 X .INT. Y = F
 X .PSB. Y = F
 X .PSP. Y = F
 X .SP.  Y = F
 X .SB.  Y = F
 X, Y = ? [1,2] [1,3]
 For X = [1.0,2.0] , and Y = [1.0,3.0]
 X .IH.  Y = [1.0,3.0]
 X .IX.  Y = [1.0,2.0]
 X .DJ.  Y = F
 R .IN.  Y = T
 X .INT. Y = F
 X .PSB. Y = T
 X .PSP. Y = F
 X .SP.  Y = F
 X .SB.  Y = T
 X, Y = ? <Control-D>
```

1.3.12 Intrinsic `INTERVAL`-Specific Functions

A variety of intrinsic `INTERVAL`-specific functions are provided. See Section 2.9.4.4 "Intrinsic Functions" on page 120. Use CODE EXAMPLE 1-13 to explore how intrinsic `INTERVAL` functions behave.

CODE EXAMPLE 1-13 Intrinsic `INTERVAL`-Specific Functions

```
math% cat ce1-13.f95
   INTERVAL ::  X, Y
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, *, IOSTAT=IOS) X
   DO WHILE (IOS >= 0)
       PRINT *, " For X =", X
       PRINT *, 'MID(X)= ', MID(X)
       PRINT *, 'MIG(X)= ', MIG(X)
       PRINT *, 'MAG(X)= ', MAG(X)
       PRINT *, 'WID(X)= ', WID(X)
       PRINT *, 'NDIGITS(X)= ', NDIGITS(X)
       WRITE(*, 1, ADVANCE='NO')
       READ(*, *, IOSTAT=IOS) X
   END DO
1  FORMAT(" X = ?")
   END
math% f95 -xia ce1-13.f95
math% a.out
 Press Control/D to terminate!
 X = ?[1.23456,1.234567890]
 For X = [1.2345599999999998,1.2345678900000002]
 MID(X)=  1.234563945
 MIG(X)=  1.2345599999999998
 MAG(X)=  1.2345678900000001
 WID(X)=  7.890000000232433E-6
 NDIGITS(X)=  6
 X = ?[1,10]
 For X = [1.0,10.0]
 MID(X)=  5.5
 MIG(X)=  1.0
 MAG(X)=  10.0
 WID(X)=  9.0
 NDIGITS(X)=  1
 X = ? <Control-D>
```

1.3.13 Interval Versions of Standard Intrinsic Functions

Every Fortran intrinsic function that accepts REAL arguments has an interval version. See Section 2.9.4.4 "Intrinsic Functions" on page 120. Use CODE EXAMPLE 1-14 to explore how some intrinsic functions behave.

CODE EXAMPLE 1-14 Interval Versions of Standard Intrinsic Functions

```
math% cat ce1-14.f95
   INTERVAL :: X, Y
   INTEGER  :: IOS = 0
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, *, IOSTAT=IOS) X
   DO WHILE (ios >= 0)
      PRINT *, "For X =", X
      PRINT *, 'ABS(X) = ', ABS(X)
      PRINT *, 'LOG(X) = ', LOG(X)
      PRINT *, 'SQRT(X)= ', SQRT(X)
      PRINT *, 'SIN(X) = ', SIN(X)
      PRINT *, 'ACOS(X)= ', ACOS(X)
      WRITE(*, 1, ADVANCE='NO')
      READ(*, *, IOSTAT=IOS) X
   END DO
1  FORMAT(" X = ?")
   END
math% f95 -xia ce1-14.f95
math% a.out
 Press Control/D to terminate!
 X = ?[1.1,1.2]
For X = [1.0999999999999998,1.2000000000000002]
 ABS(X) =  [1.0999999999999998,1.2000000000000002]
 LOG(X) =  [0.095310179804324726,0.18232155679395479]
 SQRT(X)=  [1.0488088481701514,1.0954451150103324]
 SIN(X) =  [0.89120736006143519,0.93203908596722652]
 ACOS(X)=  [EMPTY]
 X = ?[-0.5,0.5]
For X = [-0.5,0.5]
 ABS(X) =  [0.0E+0,0.5]
 LOG(X) =  [-Inf,-0.69314718055994528]
 SQRT(X)=  [0.0E+0,0.70710678118654758]
 SIN(X) =  [-0.47942553860420307,0.47942553860420307]
 ACOS(X)=  [1.0471975511965976,2.0943951023931958]
 X = ? <Control-D>
```

1.4 Code Development Tools

Information on interval code development tools is available online. See the Interval Arithmetic README for a list of interval web sites and other online resources.

To report a suspected interval error, send email to

```
sun-dp-comments@Sun.COM
```

Include the following text in the Subject line or the email message:

`WORKSHOP "6.0` *mm/dd/yy*`" Interval`

where *mm/dd/yy* is the month, day, and year.

1.4.1 Debugging Support

In Sun WorkShop 6, interval data types are supported by dbx to the following extent:

- The values of individual `INTERVAL` variables can be printed using the `print` command.
- The value of all `INTERVAL` variables can be printed using the `dump` command.
- New values can be assigned to `INTERVAL` variables using the `assign` command.
- Printing the value of `INTERVAL` expressions is not supported.
- There is no provision to visualize `INTERVAL` data arrays.
- All generic functionality that is not data type specific should work.

For additional details on `dbx` functionality, see *Debugging a Program With dbx.*

1.4.2 Global Program Checking

Global program checking (GPC) in Sun WorkShop 6 Fortran 95 detects one interval-specific error: `INTERVAL` type mismatches in user-supplied routine calls.

CODE EXAMPLE 1-15 INTERVAL Type Mismatch

```
math% cat ce1-15.f95
INTERVAL X
X = [-1.0,+2.9]
PRINT *,X
CALL SUB(X)
END
SUBROUTINE SUB(Y)
INTEGER Y(2)
PRINT *,Y
END
math% f95 -xia ce1-15.f95 -Xlist

--- See ce1-15.lst ---

              Global Call-Chain Considerata
             ===============================

     1) <503>  At line 4, MainPgm() calls SUB(fileline 6):
        MainPgm() sends argument 1 as type "Interval(16),"
        but SUB() expects type "Integer(4)"

     2) <507>  At line 4, MainPgm() calls SUB(fileline 6):
        MainPgm() sends argument 1 as a "Scalar,"
        but SUB() expects a "1-D Array"
```

1.4.3 Interval Functionality Provided in Sun Fortran Libraries

The following libraries contain intrinsic INTERVAL routines.

TABLE 1-3 Interval Libraries

Library	Name	Needed Options
intrinsic INTERVAL array functions	libifai	None
intrinsic INTERVAL library	libsunimath	None

1.4.4 Porting Code and Binary Files

There is limited legacy interval Fortran code with which to contend. Until language syntax and semantics are standardized, different providers of interval compiler support will inevitably diverge. The standardization process will be facilitated if users provide feedback regarding the most favored `INTERVAL` syntax and semantics. Comments can be sent to the email alias listed in the Interval Arithmetic README.

The representation of intervals in binary files will change as compilers supporting narrower interval systems are made available.

1.4.5 Parallelization

In this release, the `-autopar` compiler option has no effect on loops containing interval arithmetic operations. These loops are not automatically parallelized. The `-explicitpar` compiler option must be used to parallelize loops marked with explicit parallelization directives.

1.5 Error Detection

The following code samples list interval-specific error messages. Each code sample includes the error message and the sample code that produced the error.

CODE EXAMPLE 1-16 Invalid Endpoints

```
math% cat ce1-16.f95
INTERVAL :: I = [2., 1.]
END

math% f95 -xia ce1-16.f95

INTERVAL :: I = [2., 1.]
                       ^
"ce1-14.f95", Line = 1, Column = 24: ERROR: The left endpoint of
the interval constant must be less than or equal to the right
endpoint.

f90: COMPILE TIME 0.150000 SECONDS
f90: MAXIMUM FIELD LENGTH 4117346 DECIMAL WORDS
f90: 2 SOURCE LINES
f90: 1 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

CODE EXAMPLE 1-17 Equivalence of Intervals and Non-Intervals

```
math% cat ce1-17.f95
INTERVAL :: I
REAL     :: R
EQUIVALENCE (I, R)
END

math% f95 -xia ce1-17.f95

EQUIVALENCE (I, R)
             ^
"ce1-15.f95", Line = 3, Column = 14: ERROR: Equivalence of INTERVAL
object "I" and REAL object "R" is not allowed.

f90: COMPILE TIME 0.160000 SECONDS
f90: MAXIMUM FIELD LENGTH 4117346 DECIMAL WORDS
f90: 4 SOURCE LINES
f90: 1 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

CODE EXAMPLE 1-18 Equivalence of `INTERVAL` Objects with Different KTPVs

```
math% cat ce1-18.f95
INTERVAL(4) :: I1
INTERVAL(8) :: I2
EQUIVALENCE (I1, I2)
END

math% f95 -xia ce1-18.f95

EQUIVALENCE (I1, I2)
             ^
"ce1-16.f95", Line = 3, Column = 14: ERROR: Equivalence of the
interval objects "I1" and  "I2" with the different kind type
parameters is not allowed.

f90: COMPILE TIME 0.190000 SECONDS
f90: MAXIMUM FIELD LENGTH 4117346 DECIMAL WORDS
f90: 4 SOURCE LINES
f90: 1 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

CODE EXAMPLE 1-19 Assigning a REAL Expression to an INTERVAL Variable in Strict Mode

```
math% cat ce1-19.f95
INTERVAL :: X
REAL     :: R
X = R
END
math% f95 -xia=strict ce1-19.f95

X = R
  ^
"ce1-17.f95", Line = 3, Column = 3: ERROR: Assignment of a REAL
expression to a INTERVAL variable is not allowed.

f90: COMPILE TIME 0.350000 SECONDS
f90: MAXIMUM FIELD LENGTH 4117346 DECIMAL WORDS
f90: 4 SOURCE LINES
f90: 1 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

CODE EXAMPLE 1-20 Assigning an INTERVAL Expression to INTERVAL Variable in Strict Mode

```
math% cat ce1-20.f95
INTERVAL     :: X
INTERVAL(16) :: y
X = Y
END
math% f95 -xia=strict ce1-20.f95

X = Y
  ^
"ce1-18.f95", Line = 3, Column = 3: ERROR: Assignment of an
interval expression to an interval variable is not allowed when
they have different kind type parameter values.

f90: COMPILE TIME 0.170000 SECONDS
f90: MAXIMUM FIELD LENGTH 4117346 DECIMAL WORDS
f90: 4 SOURCE LINES
f90: 1 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

1.5.1 Known Containment Failures

Whenever an interval containment failure can occur, a compile-time warning should be issued. An integer expression outside the scope of widest-need expression processing is the only known situation in which such a warning is necessary.

1.5.1.1 Integer Overflow

Numerical inaccuracies are normally associated with `REAL` rather than `INTEGER` expressions. In one respect, `INTEGER` expressions are more dangerous than `REAL` expressions. When `REAL` expressions overflow, an exception is raised, and an IEEE infinity is generated. The exception is a warning that overflow has occurred. Infinities tend to propagate in floating-point computations, thereby alerting users of a potential problem. It is also possible to trap on overflow.

When `INTEGER` expressions overflow, they silently wrap around to some possibly-opposite-signed value. Moreover, the only practical way to detect integer overflow is to perform the inverse operation and test for equality on every integer operation. Integer constant expressions are safe because they are evaluated during compilation where overflow is detected and signalled with a warning message.

The following example shows what can happen if the scope of widest-need expression processing is not extended to all intrinsic `INTEGER` operations and functions, including the ** operation with an `INTEGER` exponent.

CODE EXAMPLE 1-21 `INTEGER` Overflow Containment Failure

```
math% cat ce1-21.f95
   INTERVAL :: X = [2], Y = [2]
   INTEGER  :: I = HUGE(0)
   X = X**(I+1)
   Y = Y*(Y**I)
   IF(X .DJ. Y) PRINT *, "X and Y are disjoint."
   END
math% f95 -xia ce1-21.f95
math% a.out
 X and Y are disjoint.
```

This code demonstrates a silent containment failure. It is a known error because the scope of widest need expression processing does not presently extend to the integer exponent of the ** operation. For information on the power operator, see Section 2.5 “Power Operators `X**N` and `X**Y`” on page 60.

Note – This error has not been fixed in the Sun WorkShop 6 Fortran 95 release, and no warning messages are issued.

CHAPTER 2

f95 Interval Reference

This chapter is a reference for the syntax and semantics of the intrinsic INTERVAL types implemented in Sun WorkShop 6 Fortran 95. The sections can be read in any order.

Unless explicitly stated otherwise, the INTERVAL data type has the same properties as other intrinsic numeric types. This chapter highlights differences between the REAL and INTERVAL types.

Some code examples are not complete programs. The implicit assumption is that these examples are compiled with the -xia command line option.

2.1 Fortran Extensions

INTERVAL data types are a non-standard extension to Fortran. However, where possible, the implemented syntax and semantics conform to the style of Fortran.

2.1.1 Character Set Notation

Left and right square brackets, "[...]", are added to the Fortran character set to delimit literal INTERVAL constants.

Throughout this document, unless explicitly stated otherwise, INTEGER, REAL, and INTERVAL constants mean *literal* constants. Constant expressions and named constants (PARAMETERS) are always explicitly identified as such.

TABLE 2-1 shows the character set notation used for code and mathematics.

TABLE 2-1 Font Conventions

Character Set	Notation
Fortran code	`INTERVAL :: X=[0.1,0.2]`
Input to programs and commands	`Enter X: ?` **`[2.3,2.4]`**
Placeholders for constants in code	`[`*`a`*`,`*`b`*`]`
Scalar mathematics	$x(a + b) = xa + xb$
Interval mathematics	$X(A + B) \subseteq XA + XB$

Note – Pay close attention to font usage. Different fonts represent an interval's exact, external mathematical value and an interval's machine-representable, internal approximation.

2.1.2 INTERVAL Constants

In f95, an INTERVAL constant is either a single integer or real decimal number enclosed in square brackets, [3.5], or a pair of integer or real decimal numbers separated by a comma and enclosed in square brackets, [3.5 E-10, 3.6 E-10]. If a degenerate interval is not machine representable, directed rounding is used to round the exact mathematical value to an internal machine representable interval known to satisfy the containment constraint.

An INTERVAL constant with both endpoints of type default INTEGER, default REAL or REAL(8), has the default type INTERVAL.

If an endpoint is of type default INTEGER, default REAL or REAL(8), it is internally converted to a value of the type REAL(8).

If an endpoint's type is INTEGER(8), it is internally converted to a value of type REAL(16).

If an endpoint's type is INTEGER(4), it is internally converted to a value of type REAL(8).

If an endpoint's type is INTEGER(1) or INTEGER(2), it is internally converted to a value of type REAL(4).

If both endpoints are of type REAL but have different KTPVs, they are both internally represented using the approximation method of the endpoint with greater decimal precision.

The KTPV of an INTERVAL constant is the KTPV of the part with the greatest decimal precision.

CODE EXAMPLE 2-1 shows the KTPV of various INTERVAL constants.

CODE EXAMPLE 2-1 KTPV of INTERVAL Constants

```
math% cat ce2-1.f95
IF(KIND([9_8, 9.0])      == 16 .AND. &
   KIND([9_8, 9_8])      == 16 .AND. &
   KIND([9_4, 9_4])      == 8  .AND. &
   KIND([9_2, 9_2])      == 4  .AND. &
   KIND([9, 9.0_16])     == 16 .AND. &
   KIND([9, 9.0])        == 8  .AND. &
   KIND([9, 9])          == 8  .AND. &
   KIND([9.0_4, 9.0_4])  == 4  .AND. &
   KIND([1.0Q0, 1.0_16]) == 16 .AND. &
   KIND([1.0_8, 1.0_4])  == 8  .AND. &
   KIND([1.0E0, 1.0Q0])  == 16 .AND. &
   KIND([1.0E0, 1])      == 8  .AND. &
   KIND([1.0Q0, 1])      == 16 ) PRINT *, 'Check'
END
math% f95 -xia ce2-1.f95
```

A Fortran constant, such as 0.1 or [0.1,0.2], is associated with the two values: the external value it represents and its internal approximation. In Fortran, the value of a constant is its internal approximation. There is no need to distinguish between a constant's external value and its internal approximation. Intervals require this distinction to be made. To represent a Fortran constant's external value, the following notation is used:

ev(0.1) = 0.1 or ev([0.1,0.2])= [0.1, 0.2].

The notation ev stands for *external value*.

Following the Fortran Standard, the numerical value of an INTERVAL constant is its internal approximation. The external value of an INTERVAL constant is always explicitly labelled as such.

For example, the INTERVAL constant [1, 2] and its external value ev([1, 2]) are equal to the mathematical value [1, 2]. However, while ev([0.1, 0.2]) = [0.1, 0.2], [0.1, 0.2] is only an internal machine approximation, because the numbers 0.1 and 0.2 are not machine representable. The value of the INTERVAL constant, [0.1, 0.2] is its internal machine approximation. The external value is denoted ev([0.1, 0.2]).

Under strict expression processing, an INTERVAL constant's internal approximation is fixed, as it is for other Fortran numeric typed constants. The value of a REAL constant is its internal approximation. Similarly, the value of an INTERVAL constant's internal approximation is referred to as the constant's value. A constant's external value, which is not a defined concept in standard Fortran, can be different from its internal approximation. Under widest-need expression processing, an INTERVAL constant's internal value is context-dependent. Nevertheless, an INTERVAL constant's internal approximation must contain its external value in both strict and widest-need expression processing.

Like any mathematical constant, the external value of an INTERVAL constant is invariant. The external value of a named INTERVAL constant (PARAMETER) cannot change within a program unit. However, as with any named constant, in different program units, different values can be associated with the same named constant.

Because intervals are opaque, there is no language requirement to store the information needed to internally represent an interval. Intrinsic functions are provided to access the infimum and supremum of an interval. Nevertheless, an INTERVAL constant is defined by an ordered pair of REAL or INTEGER constants. The constants are separated by a comma, and the pair is enclosed in square brackets. The first constant is the infimum or lower bound, and the second, is the supremum or upper bound.

If only one constant appears inside the square brackets, the represented interval is degenerate, having equal infimum and supremum. In this case, an internal interval approximation is constructed that is guaranteed to contain the single decimal literal constant's external value.

A valid interval must have an infimum that is less than or equal to its supremum. Similarly, an INTERVAL constant must also have an infimum that is less than or equal to its supremum. For example, the following code fragment must evaluate to *true*:

```
INF([0.1]) .LE. SUP([0.1]).
```

CODE EXAMPLE 2-2 contains examples of valid and invalid INTERVAL constants.

For additional information regarding INTERVAL constants, see the supplementary paper [4] cited in Section 2.10 "References" on page 126

CODE EXAMPLE 2-2 Valid and Invalid INTERVAL Constants

```
X=[2,3]
X=[0.1]       ! Line 2: Interval containing the decimal number 1/10
X=[2, ]       ! Line 3: Invalid - missing supremum
X=[3_2,2_2]   ! Line 4: Invalid - infimum > supremum
X=[2_8,3_8]
X=[2,3_8]
X=[0.1E0_8]
X=[2_16,3_16]
X=[2,3_16]
X=[0.1E0_16]
```

2.1.3 Internal Approximation

The internal approximation of a REAL constant does not necessarily equal the constant's external value. For example, because the decimal number 0.1 is not a member of the set of binary floating-point numbers, this value can only be *approximated* by a binary floating-point number that is close to 0.1. For REAL data items, the approximation accuracy is unspecified in the Fortran standard. For INTERVAL data items, a pair of floating-point values is used that is known to contain the set of mathematical values defined by the decimal numbers used to symbolically represent an INTERVAL constant. For example, the mathematical interval [0.1, 0.2] is the external value of the INTERVAL constant [0.1,0.2].

Just as there is no Fortran language requirement to accurately approximate REAL constants, there is also no language requirement to approximate an interval's external value with a narrow width INTERVAL constant. There is a requirement for an INTERVAL constant to *contain* its external value.

$$\mathrm{ev}(\texttt{INF([0.1,0.2])}) \le \mathrm{inf}(\mathrm{ev}(\texttt{[0.1,0.2]})) = \mathrm{inf}([0.1, 0.2])$$

and

$$\mathrm{sup}([0.1, 0.2]) = \mathrm{sup}(\mathrm{ev}(\texttt{[0.1,0.2]})) \le \mathrm{ev}(\texttt{SUP([0.1,0.2])})$$

f95 INTERVAL constants are sharp. This is a quality of implementation feature.

2.1.4 INTERVAL Statement

The INTERVAL declaration statement is the only INTERVAL-specific statement added to the Fortran language in f95. For a detailed description of the INTERVAL declaration statement and standard Fortran statements that interact with INTERVAL data items, see "INTERVAL Statements" on page 88.

2.2 Data Type and Data Items

If the -xia or -xinterval options are entered in the f95 command line, or if they are set either to widestneed or to strict, the INTERVAL data type is recognized as an intrinsic numeric data type in f95. If neither option is entered in the f95 command line, or if they are set to no, the INTERVAL data type is not recognized as intrinsic. See Section 2.3.3 "Interval Command-Line Options" on page 52 for details on the INTERVAL command-line options.

2.2.1 Name: INTERVAL

The intrinsic type INTERVAL is added to the six intrinsic Fortran data types. The INTERVAL type is opaque, meaning that an INTERVAL data item's internal format is not specified. Nevertheless, an INTERVAL data item's external format is a pair of REAL data items having the same kind type parameter value (KTPV) as the INTERVAL data item.

2.2.2 Kind Type Parameter Value (KTPV)

An INTERVAL data item is an approximation of a mathematical interval consisting of a lower bound or infimum and an upper bound or supremum. INTERVAL data items have all the properties of other numeric data items.

The KTPV of a default INTERVAL data item is 8. The size of a default INTERVAL data item with no specified KTPV is 16 bytes. The size of a default INTERVAL data item in f95 cannot be changed using the -xtypemap or -r8const command line options. For more information, see Section 2.3.3.1 "-xtypemap and -r8const Command-Line Options" on page 53. Thus

```
KIND([0])= 2*KIND(0) = KIND(0.0_8) = 8
```

provided the size of the default REAL and INTEGER data items is not changed using -xtypemap.

2.2.2.1 Size and Alignment Summary

The size and alignment of `INTERVAL` types is unaffected by `f95` compiler options. TABLE 2-2 contains `INTERVAL` sizes and alignments.

TABLE 2-2 `INTERVAL` Sizes and Alignments

Data Type	Byte Size	Alignment
`INTERVAL`	16	8
`INTERVAL(4)`	8	4
`INTERVAL(8)`	16	8
`INTERVAL(16)`	32	16

Note – `INTERVAL` arrays align the same as their elements.

2.2.3 `INTERVAL` Arrays

`INTERVAL` arrays have all the properties of arrays with different numeric types. See CODE EXAMPLE 2-25 on page 91 for the declaration of `INTERVAL` arrays.

Interval versions of the following intrinsic array functions 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()` intrinsic functions are not defined for `INTERVAL` arrays because the `MINVAL` and `MAXVAL` intrinsic applied to an `INTERVAL` array may return an interval value not possessed by any element of the array. See the following sections for descriptions of the `MAX` and `MIN` intrinsic functions

- Section 2.9.4.2 "Maximum: `MAX(X1,X2,[X3,...])`" on page 119
- Section 2.9.4.3 "Minimum: `MIN(X1,X2,[X3, ...])`" on page 119.

For example `MINVAL((/[1,2],[3,4]/)) = [1,3]` and

`MAXVAL(/[1,2],[3,4]/) = [2,4]`.

Array versions of the following intrinsic `INTERVAL`-specific functions are supported: `ABS()`, `INF()`, `INT()`, `MAG()`, `MAX()`, `MID()`, `MIG()`, `MIN()`, `NDIGITS()`, `SUP()`, `WID()`.

Array versions of the following intrinsic `INTERVAL`-mathematical functions are supported: `ACOS()`, `AINT()`, `ANINT()`, `ASIN()`, `ATAN()`, `ATAN2()`, `CEILING()`, `COS()`, `COSH()`, `EXP()`, `FLOOR()`, `LOG()`, `LOG10()`, `MOD()`, `SIGN()`, `SIN()`, `SINH()`, `SQRT()`, `TAN()`, `TANH()`.

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

2.3 `INTERVAL` Arithmetic Expressions

`INTERVAL` arithmetic expressions are constructed from the same arithmetic operators as other numerical data types. The fundamental difference between `INTERVAL` and non-`INTERVAL` (point) expressions is that the result of any possible `INTERVAL` expression is a valid `INTERVAL` that satisfies the containment constraint of interval arithmetic. In contrast, point expression results can be any approximate value.

2.3.1 Mixed-Mode `INTERVAL` Expressions

Mixed-mode (`INTERVAL`-point) expressions require widest-need expression processing to guarantee containment. Expression processing is widest-need by default when support for intervals is invoked using either the `-xia` command-line macro or the `-xinterval` command line option. If widest-need expression processing is not wanted, use the options `-xia=strict` or `-xinterval=strict` to invoke strict expression processing. Mixed-mode `INTERVAL` expressions are compile-time errors under strict expression processing. Mixed-mode operations between `INTERVAL` and `COMPLEX` operands are not supported.

With widest-need expression processing, the KTPV of all operands in an interval expression is promoted to KTPV_{max}, the highest `INTERVAL` KTPV found anywhere in the expression.

Note – KTPV promotion is performed before expression evaluation.

Widest-need expression processing guarantees:

- Interval containment
- No type or precision conversions add width to the converted intervals.

Caution – Unless there is a specific requirement to use strict expression processing, it is strongly recommended that users employ widest-need expression processing. In any expression or subexpression, explicit `INTERVAL` type and KTPV conversions can always be made.

Each of the following examples is designed to illustrate the behavior and utility of widest-need expression processing. There are three blocks of code in each example:

- Generic code that is independent of the expression processing mode (widest-need, or strict)
- Widest-need code
- Equivalent strict code

The examples are designed to communicate three messages:

- Except in special circumstances, use the widest-need expression processing.
- Whenever widest-need expression processing is enabled, but is not wanted, it can be overridden using the `INTERVAL` constructor to coerce type and KTPV conversions.
- With strict expression processing, `INTERVAL` type and precision conversions must be explicitly specified using `INTERVAL` constants and the `INTERVAL` constructor.

2.3.2 Value Assignment

The `INTERVAL` assignment statement assigns a value of an `INTERVAL` scalar, array element, or array expression to an `INTERVAL` variable, array element or array. The syntax is:

```
V = expr
```

where *`expr`* is a placeholder for an interval arithmetic or array expression, and `V` is an `INTERVAL` variable, array element, or array.

Executing an `INTERVAL` assignment causes the expression to be evaluated using either widest-need or strict expression processing. The resulting value is then assigned to `V`. The following steps occur when evaluating an expression using widest-need expression processing:

1. The interval KTPV of every point (non-`INTERVAL`) data item is computed.

   If the point item is an integer, the resulting interval KTPV is twice the integer's KTPV. Otherwise an interval's KTPV is the same as the point item's KTPV.

2. The expression, including the left-hand side of an assignment statement, is scanned for the maximum interval KTPV, denoted KTPV_{max}.

3. All point and INTERVAL data items in the INTERVAL expression are promoted to $KTPV_{max}$, prior to evaluating the expression.
4. If KIND(V) < $KTPV_{max}$, the expression result is converted to a containing interval with KTPV = KIND(V) and the resulting value is assigned to V.

CODE EXAMPLE 2-3 $KTPV_{max}$ Depends on KIND (Left-Hand Side).

```
math% cat ce2-3.f95
INTERVAL(4)  :: X1, Y1
INTERVAL     :: X2, Y2          ! Same as: INTERVAL(8) :: X2, Y2
INTERVAL(16) :: X3, Y3

! Widest-need code
 X1 = 0.1
 X2 = 0.1
 X3 = 0.1

! Equivalent strict code
Y1 = [0.1_4]
Y2 = [0.1_8]
Y3 = [0.1_16]

IF(X1 .SEQ. Y1)  PRINT *, "Check1."
IF(X2 .SEQ. Y2)  PRINT *, "Check2."
IF(X3 .SEQ. Y3)  PRINT *, "Check3."
END

math% f95 -xia ce2-3.f95
math% a.out
 Check1.
 Check2.
 Check3.
```

Note – Under widest-need, the KTPV of the variable to which assignment is made (the left-hand side) is included in determining the value of $KTPV_{max}$ to which all items in an INTERVAL statement are promoted.

CODE EXAMPLE 2-4 Mixed-Mode Assignment Statement

```
math% cat ce2-4.f95
INTERVAL(4) :: X1, Y1
INTERVAL(8) :: X2, Y2
REAL(8)     :: R = 0.1

! Widest-need code
 X1 = R*R                                                          ! Line 4
 X2 = X1*R                                                         ! Line 5

! Equivalent strict  code
 Y1 = INTERVAL((INTERVAL(R, KIND=8)*INTERVAL(R, KIND=8)), KIND=4  )! Line 6
 Y2 = INTERVAL(X1, KIND=8)*INTERVAL(R, KIND=8)                     ! Line 7

IF((X1 == Y1)) PRINT *, "Check1."                                  ! Line 8
IF((X2 == Y2)) PRINT *, "Check2."                                  ! Line 9
END

math% f95 -xia ce2-4.f95
math% a.out
 Check1.
 Check2.
```

CODE EXAMPLE 2-4 notes:

- The equivalent strict code shows the steps required to reproduce the results obtained using widest-need expression processing.
- In line 4, `KIND(R)`= 8, but `KIND(X1)` = 4. To guarantee containment and produce a sharp result, `R` is converted to a $KTPV_{max}$ = 8 containing interval before evaluating the expression. Then the result is converted to a KTPV-4 containing interval and assigned to `X1`. These steps are made explicit in the equivalent strict code in line 6.
- In line 5, `KIND(R)` = `KIND(X2)` = 8. Therefore, `X1` is promoted to a KTPV-8 `INTERVAL` before the expression is evaluated and the result assigned to `X2`. Line 7 shows the equivalent strict code.
- The checks in lines 8 and 9 verify that the widest-need and strict results are identical. For more detailed information on widest-need and strict expression processing, see Section 2.3 "`INTERVAL` Arithmetic Expressions" on page 48.

2.3.3 Interval Command-Line Options

Interval features in the `f95` compiler are activated by means of the following command-line options:

- `-xinterval=(no|widestneed|strict)` is a command-line option to enable processing of intervals and to control permitted expression evaluation syntax.
  - "`no`" disables the interval extensions to `f95`.
  - "`widestneed`" enables widest-need expression processing and functions the same as `-xinterval` if no option is specified. See Section 2.3.1 “Mixed-Mode `INTERVAL` Expressions” on page 48.
  - "`strict`" requires all `INTERVAL` type and KTPV conversions to be explicit, or it is a compile-time error, as described in Section 1.5 “Error Detection” on page 37.
- `-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 (the first default), 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 superseded.

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`

When using command-line options:

- At the end of the command-line processing, if `-ansi` is set and `-xinterval` is set to either `widestneed` or `strict`, the following warning is issued: `"Interval data types are a non-standard feature"`.
- `-fround = <r>`: (Set the IEEE rounding mode in effect at startup) does not interact with `-xia` because `INTERVAL` operations and routines save and restore the rounding mode upon entry and exit.

When recognition of INTERVAL types is activated:

- INTERVAL operators and functions become intrinsic.
- The same restrictions are imposed on the extension of intrinsic INTERVAL operators and functions as are imposed on the extension of standard intrinsic operators and functions.
- Intrinsic INTERVAL-specific function names are recognized. See Section 2.2.3 "INTERVAL Arrays" on page 47 and Section 2.9.4 "Mathematical Functions" on page 116.

2.3.3.1 `-xtypemap` and `-r8const` Command-Line Options

The size of a default INTERVAL variable declared only with the INTERVAL keyword cannot be changed using the `-xtypemap` and `-r8const` command line options.

While these command line options have no influence on the size of default INTERVAL types, the options can change the result of mixed-mode INTERVAL expressions, as shown in CODE EXAMPLE 2-5.

CODE EXAMPLE 2-5 Mixed-Mode Expression

```
math% cat ce2-5.f95
REAL     :: R
INTERVAL :: X
R = 1.0E0 - 1.0E-15
PRINT *, 'R = ', R
X = 1.0E0 - R
PRINT *, 'X = ', X
IF (  0.0 .IN. X  ) THEN
    PRINT *, 'X contains zero'
ELSE
    PRINT *, 'X does not contain zero'
ENDIF
PRINT *, 'WID(X) = ', WID(X)
END
math% f95 -xia ce2-5.f95
math% a.out
 R =  1.0
 X =  [0.0E+0,0.0E+0]
 X contains zero
 WID(X) =  0.0E+0
math% f95 -xia -xtypemap=real:64,double:64,integer:64 ce2-5.f95
math% a.out
 R =  0.999999999999999
 X =  [9.992007221626408E-16,9.992007221626409E-16]
 X does not contain zero
 WID(X) =  0.0E+0
```

Note – Although `-xtypemap` has no influence on the KTPV of `X`, it can influence the value of `X`.

2.3.4 Constant Expressions

`INTERVAL` constant expressions may contain `INTERVAL` literal and named constants, as well as any point constant expression components. Therefore, each operand or argument is itself, another constant expression, a constant, a named constant, or an intrinsic function called with constant arguments.

CODE EXAMPLE 2-6 Constant Expressions

```
math% cat ce2-6.f95
INTERVAL :: P, Q
! Widest-need code
P = SIN([1.23])+[3.45]/[9, 11.12]

! Equivalent strict code
Q = SIN([1.23_8])+[3.45_8]/[9.0_8, 11.12_8]
IF(P .SEQ. Q) PRINT *, 'Check'
END
math% f95 -xia ce2-6.f95
math% a.out
 Check
```

Note – Under widest-need expression processing, interval context is used to determine the KTPV of `INTERVAL` constants. See Section 1.3.7 "Default Kind Type Parameter Value (KTPV)" on page 23 for more information.

`INTERVAL` constant expressions are permitted wherever an `INTERVAL` constant is allowed.

2.4 Intrinsic Operators

TABLE 2-3 lists the intrinsic operators that can be used with intervals. In TABLE 2-3, X and Y are intervals.

TABLE 2-3 INTRINSIC Operators

Operator	Operation	Expression	Meaning
`**`	Exponentiation	`X**Y`	Raise X to the INTERVAL power Y
		`X**N`	Raise X to the INTEGER power N (See Note 1)
`*`	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

1. If N is an integer expression, overflow can cause a containment failure. Users must be responsible for preventing integer overflow in this release. See Section 1.5.1.1 “Integer Overflow” on page 40 for more information.

Precedence of operators:

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

With the exception of the interval ** operator and an integer exponent, interval operators can only be applied to two interval operands with the same kind type parameter value. Thus the type and KTPV of an interval operator’s result are the same as the type and KTPV of its operands.

If the second operand of the interval ** operator is an integer, the first operand can be of any interval KTPV. In this case, the result has the type and KTPV of the first operand.

Some INTERVAL-specific operators have no point analogs. These can be grouped into three categories: set, certainly, and possibly, as shown in TABLE 2-4. A number of unique set-operators have no certainly or possibly analogs.

TABLE 2-4 Intrinsic INTERVAL Relational Operators

Set Relational Operators	.SP.	.PSP	.SB.	.PSB.	.IN.	.DJ.
	.EQ. (same as ==)	.NEQ. (same as /=)				
	.SEQ.	.SNE.	.SLT.	.SLE.	.SGT.	.SGE.
Certainly Relational Operators	.CEQ.	.CNE.	.CLT.	.CLE.	.CGT.	.CGE.
Possibly Relational Operators	.PEQ.	.PNE.	.PLT.	.PLE.	.PGT.	.PGE.

The precedence of intrinsic INTERVAL relational operators is the same as the precedence of REAL relational operators.

Except for the .IN. operator, intrinsic INTERVAL relational operators can only be applied to two INTERVAL operands with the same KTPV.

The first operand of the .IN. operator is of any INTEGER or REAL type. The second operand can have any interval KTPV.

The result of the INTERVAL relational expression has the default LOGICAL kind type parameter.

2.4.1 Arithmetic Operators +, –, *, /

Formulas for computing the endpoints of interval arithmetic operations on finite REAL intervals are motivated by the requirement to produce the narrowest interval that is guaranteed to contain the set of all possible point results. Ramon Moore independently developed these formulas and more importantly, was the first to develop the analysis needed to apply interval arithmetic. For more information, see R. Moore, *Interval Analysis*, Prentice-Hall, 1966.

The set of all possible values was originally defined by performing the operation in question on any element of the operand intervals. Therefore, given finite intervals, [a, b] and [c, d], with $\mathrm{op} \in \{+, -, \times, \div\}$,

$$[a, b]\ \mathrm{op}\ [c, d] \supseteq \{x\ \mathrm{op}\ y \mid x \in [a, b] \text{ and } y \in [c, d]\},$$

with division by zero being excluded. The formulas, or their logical equivalent, are:

$$[a, b] + [c, d] = [a + c\ ,\ b + d]$$

$$[a, b] - [c, d] = [a - d\ ,\ b - c]$$

$$[a, b] \times [c, d] = [\min(a \times c, a \times d, b \times c, b \times d)\ ,\ \max(a \times c, a \times d, b \times c, b \times d)]$$

$$[a, b] / [c, d] = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right], \text{ if } 0 \notin [c, d]$$

Directed rounding is used when computing with finite precision arithmetic to guarantee the set of all possible values is contained in the resulting interval.

The set of values that any interval result must contain is called the containment set of the operation or expression that produces the result.

To include extended intervals (with infinite endpoints) and division by zero, containment sets cannot directly depend on the value of arithmetic operations on real values. For extended intervals, containment sets are required for operations on points that are normally undefined. Undefined operations include the indeterminate forms $1 \div 0, 0 \times \infty, 0 \div 0$, and $\infty \div \infty$.

The containment-set closure identity solves the problem of identifying the value of containment sets of expressions at singular or indeterminate points. The identity states that containment sets are closures. The closure of a function at a point on the boundary of its domain includes all limit or accumulation points. For details, see the Glossary and the supplementary papers [1], [3], [10], and [11] cited in Section 2.10 "References" on page 126.

Symbolically, cset(x op y, $\{(x_0, y_0)\}$) = $\{x_0\}\ \overline{\mathrm{op}}\ \{y_0\}$, where $\overline{\mathrm{op}}$ denotes the closure of the operation, op, and $\{x_0\}$ denotes the singleton set with only one element, x_0. The subscript 0 is used to symbolically distinguish a particular value, x_0, of the variable, x, from the variable itself. For example, with $x_0 = 1$, op = ÷, and $y_0 = 0$, $x_0 \div y_0$ is undefined, but the closure, $\{1\}\ \overline{\div}\ \{0\} = \{-\infty, +\infty\}$.

This result is obtained using the sequences

$$\{y_j\} = \left\{\frac{-1}{j}\right\} \text{ or } \{y_j\} = \left\{\frac{1}{j}\right\},$$

both of whose limits are

$$\lim_{j \to \infty} y_j = 0$$

Using the two sequences, $\{y_j\}$, above, the closure of the division operator at $x_0 = 1$ and $y_0 = 0$ is:

$$\begin{aligned} 1 \mathbin{\overline{\div}} 0 &= \lim_{j \to \infty} \frac{1}{y_j} \\ &= \lim_{j \to \infty} j \text{ or } \lim_{j \to \infty} (-j) \\ &= \{-\infty, +\infty\} \end{aligned}$$

The following tables, TABLE 2-5 through TABLE 2-9, display containment sets for the four basic arithmetic operations.

TABLE 2-5 Containment Set for Addition: cset($x + y$, $\{(x_0, y_0)\}$)

cset(x + y, {(x$_0$, y$_0$)})	**{-∞}**	**{real: y$_0$}**	**{+∞}**
{-∞}	{-∞}	{-∞}	$\Re^*$
{real: x_0}	{-∞}	$\{x_0 + y_0\}$	{+∞}
{+∞}	$\Re^*$	{+∞}	{+∞}

TABLE 2-6 Containment Set for Subtraction: cset($x - y$, $\{(x_0, y_0)\}$)

cset(x – y, {(x$_0$, y$_0$)})	**{-∞}**	**{real: y$_0$}**	**{+∞}**
{-∞}	$\Re^*$	{-∞}	{-∞}
{real: x_0}	{+∞}	$\{x_0 - y_0\}$	{-∞}
{+∞}	{+∞}	{+∞}	$\Re^*$

TABLE 2-7 Containment Set for Multiplication: cset($x \times y$, $\{(x_0, y_0)\}$)

cset(x × y, {(x$_0$, y$_0$)})	**{-∞}**	**{real: y$_0$ < 0}**	**{0}**	**{real: y$_0$ > 0}**	**{+∞}**
{-∞}	{+∞}	{+∞}	$\Re^*$	{-∞}	{-∞}
{real: x$_0$ < 0}	{+∞}	$\{x \times y\}$	{0}	$\{x \times y\}$	{-∞}
{0}	$\Re^*$	{0}	{0}	{0}	$\Re^*$
{real: x$_0$ > 0}	{-∞}	$x \times y$	{0}	$x \times y$	{+∞}
{+∞}	{-∞}	{-∞}	$\Re^*$	{+∞}	{+∞}

TABLE 2-8 Containment Set for Division: cset($x \div y$, $\{(x_0, y_0)\}$)

cset($x \div y$, $\{(x_0, y_0)\}$)	$\{-\infty\}$	{real: $y_0 < 0$}	$\{0\}$	{real: $y_0 > 0$}	$\{+\infty\}$
$\{-\infty\}$	$[0, +\infty]$	$\{+\infty\}$	$\{-\infty, +\infty\}$	$\{-\infty\}$	$[-\infty, 0]$
{real: $x_0 \neq 0$}	$\{0\}$	$\{x \div y\}$	$\{-\infty, +\infty\}$	$\{x \div y\}$	$\{0\}$
$\{0\}$	$\{0\}$	$\{0\}$	$\Re^*$	$\{0\}$	$\{0\}$
$\{+\infty\}$	$[-\infty, 0]$	$\{-\infty\}$	$\{-\infty, +\infty\}$	$\{+\infty\}$	$[0, +\infty]$

All inputs in the tables are shown as singletons. Results are shown as singletons, sets, or intervals. To avoid ambiguity, customary notation, such as $(-\infty) + (+\infty) = -\infty$, $(-\infty) + y = -\infty$, and $(-\infty) + (+\infty) = \Re^*$, is not used. These tables show the results for singleton-set inputs to each operation. Results for general set (or interval) inputs are the union of the results of the single-point results as they range over the input sets (or intervals).

In one case, division by zero, the result is not an interval, but the set, $\{-\infty, +\infty\}$. In this case, the narrowest interval in the current system that does not violate the containment constraint of interval arithmetic is the interval $[-\infty, +\infty] = \Re^*$.

Sign changes produce the expected results.

To incorporate these results into the formulas for computing interval endpoints, it is only necessary to identify the desired endpoint, which is also encoded in the rounding direction. Using $\downarrow$ to denote rounding down (towards -∞) and $\uparrow$ to denote rounding up (towards +∞),

$$\downarrow(+\infty) \div (+\infty) = 0 \text{ and } \uparrow(+\infty) \div (+\infty) = +\infty.$$

$$\downarrow\ 0 \times (+\infty) = -\infty \quad \text{and } \uparrow\ 0 \times (+\infty) = +\infty.$$

Similarly, because $\text{hull}(\{-\infty, +\infty\}) = [-\infty, +\infty]$,

$$\downarrow x \div 0 = -\infty \text{ and } \uparrow x \div 0 = +\infty.$$

Finally, the empty interval is represented in Fortran by the character string `[empty]` and has the same properties as the empty set, denoted $\varnothing$ in the algebra of sets. Any arithmetic operation on an empty interval produces an empty interval result. For additional information regarding the use of empty intervals, see the supplementary papers [6] and [7] cited in Section 2.10 “References” on page 126.

Using these results, `f95` implements the closed interval system. The system is closed because all arithmetic operations and functions always produce valid interval results. See the supplementary papers [2] and [8] cited in Section 2.10 “References” on page 126.

2.5 Power Operators X**N and X**Y

The power operator can be used with integer exponents (X**N) and continuous exponents (X**Y). With a continuous exponent, the power operator has indeterminate forms, similar to the four arithmetic operators.

In the integer exponents case, the set of all values that an enclosure of X^n must contain is $\text{cset}(x^n,\{x\}) = \{z \mid z \in \text{cset}(x^n,\{x\}) \text{ and } x \in X\}$.

Monotonicity can be used to construct a sharp interval enclosure of the integer power function. When $n = 0$, cset(x_n, $\{x_0\}$)= 1 for all $x \in [-\infty, +\infty]$, and [empty]**N = [empty] for all N.

In the continuous exponents case, the set of all values that an interval enclosure of X^Y must contain is

$$\text{cset}(\exp(Y\ln(X)), \{(Y_0, X_0)\}) = \{z \mid z \in \text{cset}(\exp(y\ln(x)), \{(y_0, x_0)\}), y \in Y_0, x \in X_0\}$$

where $\text{cset}(\exp(y\ln(x)), \{(Y_0, X_0)\})$ is the containment set of the expression $\exp(y\ln(x))$. The function exp(y ln (x)) makes explicit that only values of $x \geq 0$ need be considered, and is consistent with the definition of X**Y with REAL arguments in Fortran.

The result is empty if either INTERVAL argument is empty, or if $x < 0$. This is also consistent with the point version of X**Y in Fortran.

TABLE 2-9 displays the containment sets for all the singularities and indeterminate forms of cset(exp($y$ln(x)), $\{(y_0, x_0)\}$).

TABLE 2-9 cset(exp($y$ln(x)), $\{(y_0, x_0)\}$)

x_0	y_0	**cset(exp($y$ln(x)), $\{(y_0, x_0)\}$**
0	$y_0 < 0$	$+\infty$
1	$-\infty$	$[0,+\infty]$
1	$+\infty$	$[0,+\infty]$
$+\infty$	0	$[0,+\infty]$
0	0	$[0,+\infty]$

The results in TABLE 2-9 can be obtained in two ways:

- Directly computing the closure of the composite expression, $\exp(y\ln(x))$ for the values of x_0 and y_0 for which the expression is undefined.
- Use the containment-set evaluation theorem to bound the set of values in a containment set.

For most compositions, the second option is much easier. If sufficient conditions are satisfied, the closure of a composition can be computed from the composition of its closures. That is, the closure of each sub-expression can be used to compute the closure of the entire expression. In the present case,

$$\text{cset}(\exp(y\ln(x)), \{x_0, y_0\}) = \overline{\exp}(\{y_0\} \mathbin{\overline{\times}} \overline{\ln}(\{x_0\})).$$

It is always the case that

$$\text{cset}(\exp(y\ln(x)), \{x_0, y_0\}) \subseteq \overline{\exp}(\{y_0\} \mathbin{\overline{\times}} \overline{\ln}(\{x_0\})).$$

Note that this is exactly how interval arithmetic works on intervals. The needed closures of the ln and exp functions are:

$$\begin{aligned} \overline{\ln}\{0\}\} &= \{-\infty\} \\ \overline{\ln}(\{+\infty\}) &= \{+\infty\} \\ \overline{\exp}(\{-\infty\}) &= \{0\} \\ \overline{\exp}(\{+\infty\}) &= \{+\infty\} \end{aligned}$$

A necessary condition for closure-composition equality is that the expression must be a *single-use expression* (or SUE), which means that each independent variable can appear only once in the expression.

In the present case, the expression is clearly a SUE.

The entries in TABLE 2-9 follow directly from using the containment set of the basic multiply operation in TABLE 2-7 on the closures of the ln and exp functions. For example, with $x_0 = 1$ and $y_0 = -\infty$, $\ln(x_0) = 0$. For the closure of multiplication on the values $-\infty$ and 0 in TABLE 2-7 on page 58, the result is $[-\infty, +\infty]$. Finally, $\exp([-\infty, +\infty]) = [0, +\infty]$, the second entry in TABLE 2-9. Remaining entries are obtained using the same steps. These same results are obtained from the direct derivation of the containment set of $\exp(y\ln(x))$. At this time, sufficient conditions for closure-composition equality of any expression have not been identified. Nevertheless,

- The containment-set evaluation theorem guarantees that a containment failure can never result from computing a composition of closures instead of a closure.
- An expression must be a SUE for closure-composition equality to be true.

2.6 Set Theoretic Operators

`f95` supports the following set theoretic operators for determining the interval hull and intersection of two intervals.

2.6.1 Hull: $X \underline{\cup} Y$ or `(X.IH.Y)`

Description: Interval hull of two intervals. The interval hull is the smallest interval that contains all the elements of the operand intervals.

Mathematical definitions:

$$\begin{aligned} X \texttt{ .IH. } Y &\equiv [\inf(X \cup Y), \sup(X \cup Y)] \\ &= \begin{cases} Y, \text{ if } X = \varnothing, \\ X, \text{ if } Y = \varnothing, \text{ and} \\ [\min(\underline{x}, \underline{y}), \max(\bar{x}, \bar{y})], \text{ otherwise.} \end{cases} \end{aligned}$$

Arguments: `X` and `Y` must be intervals with the same KTPV.

Result type: Same as `X`.

2.6.2 Intersection: $X \cap Y$ or `(X.IX.Y)`

Description: Intersection of two intervals.

Mathematical and operational definitions:

$$\begin{aligned} X \texttt{ .IX. } Y &\equiv \{z \mid z \in X \text{ and } z \in Y\} \\ &= \begin{cases} \varnothing, \text{ if } (X = \varnothing) \text{ or } (Y = \varnothing) \text{ or } (\min(\bar{x}, \bar{y}) < \max(\underline{x}, \underline{y})) \\ [\max(\underline{x}, \underline{y}), \min(\bar{x}, \bar{y})], \text{ otherwise.} \end{cases} \end{aligned}$$

Arguments: `X` and `Y` must be intervals with the same KTPV.

Result type: Same as `X`.

2.7 Set Relations

f95 provides the following set relations that have been extended to support intervals.

2.7.1 Disjoint: $X \cap Y = \emptyset$ or `(X .DJ. Y)`

Description: Test if two intervals are disjoint.

Mathematical and operational definitions:

$$\begin{aligned} X \texttt{ .DJ. } Y &\equiv (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or} \\ &\quad ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \forall y \in Y : x \neq y)) \\ &= (X = \emptyset) \text{ or } (Y = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and} \\ &\quad (Y \neq \emptyset) \text{ and } ((\bar{y} < \underline{x}) \text{ or } (\bar{x} < \underline{y}))) \end{aligned}$$

Arguments: X and `Y` must be intervals with the same KTPV.

Result type: Default logical scalar.

2.7.2 Element: $r \in Y$ or `(R.IN. Y)`

Description: Test if the number, `R`, is an element of the interval, `Y`.

Mathematical and operational definitions:

$$\begin{aligned} r \in Y &\equiv (\exists\, y \in Y : y = r) \\ &= (Y \neq \emptyset) \text{ and } (\underline{y} \leq r) \text{ and } (r \leq \bar{y}) \end{aligned}$$

Arguments: The type of `R` is `INTEGER` or `REAL,` and the type of `Y` is `INTERVAL`.

Result type: Default logical scalar.

The following comments refer to the $r \in Y$ set relation:

- Under widest-need expression processing, `R` and `Y` having different KTPVs has no impact on how they are evaluated. Widest-need expression processing applies to `Y`, but does not apply to the evaluation of `R`. After evaluation, KTPV promotion of `Y` or `R` is done before the inclusion test is performed.
- Under strict expression evaluation, `R` and `Y` must have the same KTPV.
- If `R` is `NaN` (Not a Number), `R .IN. Y` is unconditionally *false*.
- If `Y` is empty, `R .IN. Y` is unconditionally *false*.

2.7.3 Interior: `(X .INT. Y)`

Description: Test if `X` is in interior of `Y`.

The interior of a set in topological space is the union of all open subsets of the set.

For intervals, the relation `X .INT. Y` (X in interior of Y) means that X is a subset of Y, and both of the following relations are *false:*

- $\inf(Y) \in X$, or in Fortran: `INF(Y) .IN. X`
- $\sup(Y) \in X$, or in Fortran: `SUP(Y) .IN. X`

Note also that, $\emptyset \notin \emptyset$, but `[empty] .INT. [empty]` = *true*

The empty set is open and therefore is a subset of the interior of itself.

Mathematical and operational definitions:

$$\begin{aligned} X \;\texttt{.INT.}\; Y \equiv\ & (X = \emptyset) \text{ or} \\ & ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \exists y' \in Y, \exists y'' \in Y : y' < x < y'')) \\ =\ & (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} < \underline{x}) \text{ and } (\bar{x} < \bar{y})) \end{aligned}$$

Arguments: `X` and `Y` must be intervals with the same KTPV.

Result type: Default logical scalar.

2.7.4 Proper Subset: $X \subset Y$ or `(X .PSB. Y)`

Description: Test if X is a proper subset of Y

Mathematical and operational definitions:

$$
\begin{aligned}
X\ \texttt{.PSB.}\ Y \equiv (X \subseteq Y) \text{ and } (X \neq Y) \\
= ((X = \emptyset) \text{ and } (Y \neq \emptyset)) \text{ or } \\
(X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} < \underline{x}) \text{ and } (\bar{x} < \bar{y}) \text{ or } \\
(\underline{y} < \underline{x}) \text{ and } (\bar{x} \leq \bar{y})
\end{aligned}
$$

Arguments: X and Y must be intervals with the same KTPV.

Result type: Default logical scalar.

2.7.5 Proper Superset: $X \supset Y$ or `(X .PSP. Y)`

Description: See proper subset with $X \leftrightarrow Y$.

2.7.6 Subset: $X \subseteq Y$ or `(X .SB. Y)`

Description: Test if X is a subset of Y

Mathematical and operational definitions:

$$
\begin{aligned}
X\ \texttt{.SB.}\ Y \equiv (X = \emptyset) \text{ or } \\
((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\forall x \in X, \exists y' \in Y, \exists y'' \in Y : y' \leq x \leq y'')) \\
= (X = \emptyset) \text{ or } ((X \neq \emptyset) \text{ and } (Y \neq \emptyset) \text{ and } (\underline{y} \leq \underline{x}) \text{ and } (\bar{x} \leq \bar{y}))
\end{aligned}
$$

Arguments: X and Y must be intervals with the same KTPV.

Result type: Default logical scalar.

2.7.7 Superset: $X \supseteq Y$ or `(X.SP.Y)`

Description: See subset with $X \leftrightarrow Y$.

2.7.8 Relational Operators

An intrinsic `INTERVAL` relational operator, denoted `.`*`qop`*`.`, is composed by concatenating:

- The required period delimiters
- An operator prefix, $q \in$ {`C,P,S`}, where `C`, `P`, and `S` stand for certainly, possibly, and set, respectively
- A Fortran relational operator suffix, $op \in$ {`LT, LE`, `EQ`, `NE`, `GT`, `GE`}

In place of `.SEQ.` and `.SNE.`, `.EQ.` (or `==`) and `.NE.` (or `/=`) defaults are accepted. To eliminate code ambiguity, all other `INTERVAL` relational operators must be made explicit by specifying a prefix.

All `INTERVAL` relational operators have equal precedence. Arithmetic operators have higher precedence than relational operators.

`INTERVAL` relational expressions are evaluated by first evaluating the two operands, after which the two expression values are compared. If the specified relationship holds, the result value is *true*; otherwise, it is *false*.

When widest-need expression processing is invoked, it applies to both `INTERVAL` operand expressions of `INTERVAL` relational operators.

Letting "*nop*" stand for the complement of the operator *op*, the certainly and possibly operators are related as follows:

`.C`*`op`*`.` ≡ `.NOT.(.P`*`nop`*`.)`

`.P`*`op`*`.` ≡ `.NOT.(.C`*`nop`*`.)`

Caution – This identity between certainly and possibly operators holds unconditionally if $op \in$ {`EQ`, `NE`}, and otherwise, only if neither operand is empty. Conversely, the identity does not hold if $op \in$ {`LT`, `LE`, `GT`, `GE`} and either operand is empty.

Assuming neither operand is empty, TABLE 2-10 contains the Fortran operational definitions of all `INTERVAL` relational operators of the form:

$[\underline{x}, \overline{x}]$.*qop*.$[\underline{y}, \overline{y}]$.

The first column contains the value of the prefix, and the first row contains the value of the operator suffix. If the tabled condition holds, the result is *true*.

TABLE 2-10 Operational Definitions of Interval Order Relations

	LT.	LE.	EQ.	GE.	GT.	NE.
.S	$\underline{x} < \underline{y}$ *and* $\overline{x} < \overline{y}$	$\underline{x} \le \underline{y}$ *and* $\overline{x} \le \overline{y}$	$\underline{x} = \underline{y}$ *and* $\overline{x} = \overline{y}$	$\underline{x} \ge \underline{y}$ *and* $\overline{x} \ge \overline{y}$	$\underline{x} > \underline{y}$ *and* $\overline{x} > \overline{y}$	$\underline{x} \ne \underline{y}$ *or* $\overline{x} \ne \overline{y}$
.C	$\overline{x} < \underline{y}$	$\overline{x} \le \underline{y}$	$\overline{y} \le \underline{x}$ *and* $\overline{x} \le \underline{y}$	$\underline{x} \ge \overline{y}$	$\underline{x} > \overline{y}$	$\underline{x} > \overline{y}$ *or* $\underline{y} > \overline{x}$
.P	$\underline{x} < \overline{y}$	$\underline{x} \le \overline{y}$	$\underline{x} \le \overline{y}$ *and* $\underline{y} \le \overline{x}$	$\overline{x} \ge \underline{y}$	$\overline{x} > \underline{y}$	$\overline{y} > \underline{x}$ *or* $\overline{x} > \underline{y}$

CODE EXAMPLE 2-7 Relational Operators

```
math% cat ce2-7.f95
INTERVAL :: X = [1.0, 3.0], Y = [2.0, 4.0], Z
INTEGER  :: V = 4, W = 5
LOGICAL  :: L1, L2, L3, L4
REAL :: R

L1 = (X == X) .AND. (Y .SEQ. Y)
L2 = X .SLT. Y

! Widest-need code
Z  = W
L3 = W .CEQ. Z
L4 = X-Y .PLT. V-W
IF( L1 .AND. L2 .AND. L3 .AND. L4) PRINT *, 'Check1'

! Equivalent (for the assignment to L3 and L4) strict code
L3 = INTERVAL(W, KIND=8) .CEQ. Z
L4 = X-Y  .PLT. INTERVAL(V, KIND=8)-INTERVAL(W, KIND=8)
IF(L3 .AND. L4) PRINT *, 'Check2'
END
math% f95 -xia ce2-7.f95
math% a.out
 Check1
 Check2
```

CODE EXAMPLE 2-7 notes:

- `L1` is *true* because an interval is set-equal to itself and the default `.EQ.` (or ==) operator is the same as `.SEQ.` .
- `L2` is *true* because `(INF(X).LT.INF(Y)).AND.(SUP(X).LT.SUP(Y))` is *true*.
- `L3` is *true* because widest need promotes `W` to the interval `[5,5]` and two intervals are certainly equal if and only if all four of their endpoints are equal.
- `L4` is *true* because evaluating the interval expressions `X–Y` and `V–W` yields the intervals `[-3,1]` and `[-1,-1]` respectively. Therefore the expression `(INF(X-Y) .LT. SUP(V-W))` is *true*.

2.7.8.1 Set Relational Operators

For an affirmative order relation with

op ∈ {LT, LE, EQ, GE, GT} and

$\mathrm{op} \in \{ <, \le, =, \ge, > \}$,

between two points x and y, the mathematical definition of the corresponding set-relation, .S*op*., between two non-empty intervals X and Y is:

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

For the relation $\neq$ between two points x and y, the corresponding set relation, `.SNE.`, between two non-empty intervals X and Y is:

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

Empty intervals are explicitly considered in each of the following relations. In each case:

Arguments: `X` and `Y` must be intervals with the same KTPV.

Result type: default logical scalar.

2.7.8.2 Certainly Relational Operators

The certainly relational operators are true if the underlying relation is true for every element of the operand intervals. For example, `[a,b] .CLT. [c,d]` is true if $x < y$ for all $x \in [a, b]$ and $y \in [c, d]$. This is equivalent to $b < c$.

For an affirmative order relation with

op ∈ {LT, LE, EQ, GE, GT} and

$\mathrm{op} \in \{ <, \le, =, \ge, > \}$, between two points x and y, the corresponding certainly-*true* relation `.Cop.` between two intervals, `X` and `Y`, is

$$X\ \texttt{.C}op\texttt{.}\ Y \equiv (X \neq \varnothing) \text{ and } (Y \neq \varnothing) \text{ and } (\forall x \in X, \forall y \in Y : x \text{ op } y)\ .$$

With the exception of the anti-affirmative certainly-not-equal relation, if either operand of a certainly relation is empty, the result is *false*. The one exception is the certainly-not-equal relation, `.CNE.`, which is *true* in this case.

For each of the certainly relational operators:

Arguments: `X` and `Y` must be intervals with the same KTPV.

Result type: default logical scalar.

2.7.8.3 Possibly Relational Operators

The possibly relational operators are true if any element of the operand intervals satisfy the underlying relation. For example, `[a,b] .PLT. [c,d]` is true if there exists an $x \in [a, b]$ and a $y \in [c, d]$ such that $x < y$. This is equivalent to $a < d$.

For an affirmative order relation with

$op \in$ {`LT`, `LE`, `EQ`, `GE`, `GT`} and

$$\text{op} \in \{\ <,\ \leq,\ =,\ \geq,\ >\ \}\quad ,$$

between two points x and y, the corresponding possibly-true relation *.Pop.* between two intervals X and Y is defined as follows:

$$X\ \texttt{.P}op\texttt{.}\ Y \equiv (X \neq \varnothing) \text{ and } (Y \neq \varnothing) \text{ and } (\exists x \in X, \exists y \in Y : x \text{ op } y\ )\ .$$

If the empty interval is an operand of a possibly relation then the result is *false*. The one exception is the anti-affirmative possibly-not-equal relation, `.PNE.`, which is *true* in this case.

For each of the possibly relational operators:

Arguments: `X` and `Y` must be `INTERVALS` with the same KTPV.

Result type: default logical scalar.

2.8 Extending Intrinsic INTERVAL Operators

If the operator specified in the INTERFACE statement of a user provided operator interface block is an intrinsic INTERVAL operator (for example .IH.), an extension to the intrinsic INTERVAL operator is created.

A user-provided operator function that extends an intrinsic INTERVAL operator may not extend the operator for those data types of its operands for which this operator is predefined.

For the combinations of arguments listed below, intrinsic interval 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 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)

In CODE EXAMPLE 2-8, both S1 and S2 interfaces are correct, because .IH. is not predefined for (LOGICAL, INTERVAL(16)) operands.

CODE EXAMPLE 2-8 Interval `.IH.` Operator Extension

```
math% cat ce2-8.f95
MODULE M
INTERFACE OPERATOR (.IH.)
    MODULE PROCEDURE S1
    MODULE PROCEDURE S2
END INTERFACE
CONTAINS
REAL FUNCTION S1(L, Y)
LOGICAL, INTENT(IN)      ::  L
INTERVAL(16), INTENT(IN) ::  Y
    S1 = 1.0
END FUNCTION S1

INTERVAL FUNCTION S2(R1, R2)
REAL, INTENT(IN) ::  R1
REAL, INTENT(IN) ::  R2
    S2 = [2.0]
END FUNCTION S2
END MODULE M

PROGRAM TEST
USE M
INTERVAL(16) :: X = [1, 2]
LOGICAL      :: L = .TRUE.
REAL         :: R = 0.1
PRINT *, 'L  .IH. X  = ', L  .IH. X
PRINT *, 'R1 .IH. R2 =', R1 .IH. R2
END PROGRAM TEST

math% f95 -xia ce2-8.f95
math% a.out
 L  .IH. X  =  1.0
 R1 .IH. R2 = [2.0,2.0]
```

The extension of the + operator in CODE EXAMPLE 2-9 is incorrect because the attempt is made to change the definition of the intrinsic `INTERVAL` (+) operator, which is predefined for (`INTERVAL`, `INTERVAL`) type operands.

CODE EXAMPLE 2-9 User-defined interface that conflicts with the use of the intrinsic `INTERVAL` (+) operator.

```
math% cat ce2-9.f95
MODULE M1
INTERFACE OPERATOR (+)
    MODULE PROCEDURE S4
END INTERFACE
CONTAINS
REAL FUNCTION S4(X, Y)
INTERVAL, INTENT(IN) ::  X
INTERVAL, INTENT(IN) ::  Y
    S4 = 4.0
END FUNCTION S4
END MODULE M1

PROGRAM TEST
USE M1
INTERVAL :: X = [1.0], Y = [2.0]
PRINT *, 'X + Y = ', X + Y
END PROGRAM TEST

math% f95 -xia ce2-9.f95

MODULE M1
       ^
"ce2-9.f95", Line = 1, Column = 8: ERROR: The compiler has detected
errors in module "M1".  No module information file will be created
for this module.

    MODULE PROCEDURE S4
                     ^
"ce2-9.f95", Line = 3, Column = 22: ERROR: This specific interface
"S4" conflicts with the intrinsic use of "+".

USE M1
    ^
"ce2-9.f95", Line = 14, Column = 5: ERROR: Module "M1" has compile
errors, therefore declarations obtained from the module via the
USE statement may be incomplete.

f90: COMPILE TIME 0.820000 SECONDS
f90: MAXIMUM FIELD LENGTH 5518744 DECIMAL WORDS
f90: 17 SOURCE LINES
```

In CODE EXAMPLE 2-10, the following `S1` interface is incorrect, because `.IH.` is predefined for `(INTERVAL(4), INTERVAL(8))` operands.

CODE EXAMPLE 2-10 User-defined interface conflicts with the intrinsic use of `.IH.`.

```
math% cat ce2-10.f95
MODULE M
INTERFACE OPERATOR (.IH.)
    MODULE PROCEDURE S1
END INTERFACE
CONTAINS
INTERVAL FUNCTION S1(X, Y)
INTERVAL(4), INTENT(IN) ::  X
INTERVAL(8), INTENT(IN) ::  Y
    S1 = [1.0]
END FUNCTION S1
END MODULE M

PROGRAM TEST
USE M
INTERVAL(4) :: X = [1.0]
INTERVAL(8) :: Y = [2.0]
PRINT *, ’X .IH. Y = ’, X .IH. Y
END PROGRAM TEST
math% f95 -xia ce2-10.f95

MODULE M
       ^
"ce2-10.f95", Line = 1, Column = 8: ERROR: The compiler has
detected errors in module "M".  No module information file will be
created for this module.

    MODULE PROCEDURE S1
                     ^
"ce2-10.f95", Line = 3, Column = 22: ERROR: This specific interface
"S1" conflicts with the intrinsic use of "ih".

USE M
    ^
"ce2-10.f95", Line = 14, Column = 5: ERROR: Module "M" has compile
errors, therefore declarations obtained from the module via the
USE statement may be incomplete.

f90: COMPILE TIME 0.190000 SECONDS
f90: MAXIMUM FIELD LENGTH 4135778 DECIMAL WORDS
f90: 18 SOURCE LINES
f90: 3 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

The number of arguments of an operator function that extends an intrinsic INTERVAL operator must agree with the number of operands needed for the intrinsic operator, as shown in CODE EXAMPLE 2-11.

CODE EXAMPLE 2-11 Incorrect change in the number of arguments in a predefined INTERVAL operator.

```
math% cat ce2-11.f95
MODULE M
INTERFACE OPERATOR (.IH.)
    MODULE PROCEDURE S1
END INTERFACE
CONTAINS
REAL FUNCTION S1(R)
REAL, INTENT(IN) :: R
    S1 = 1.0
END FUNCTION S1
END MODULE M

PROGRAM TEST
USE M
REAL :: R = 0.1
PRINT *, ' .IH. R = ', .IH. R
END PROGRAM TEST
math% f95 -xia ce2-11.f95

MODULE M
       ^
"ce2-11.f95", Line = 1, Column = 8: ERROR: The compiler has
detected errors in module "M".  No module information file will be
created for this module.

    MODULE PROCEDURE S1
                     ^
"ce2-11.f95", Line = 3, Column = 22: ERROR: The specific interface
"S1" must have exactly two dummy arguments when inside a defined
binary operator interface block.

USE M
    ^
"ce2-11.f95", Line = 13, Column = 5: ERROR: Module "M" has compile
errors, therefore declarations obtained from the module via the
USE statement may be incomplete.
```

CODE EXAMPLE 2-11 Incorrect change in the number of arguments in a predefined INTERVAL operator. *(Continued)*

```
PRINT *, ' .IH. R = ', .IH. R
                       ^
"ce2-11.f95", Line = 15, Column = 24: ERROR: Unexpected syntax:
"operand" was expected but found ".".

f90: COMPILE TIME 0.200000 SECONDS
f90: MAXIMUM FIELD LENGTH 4135778 DECIMAL WORDS
f90: 16 SOURCE LINES
f90: 4 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

A binary intrinsic INTERVAL operator cannot be extended with unary operator function having an INTERVAL argument.

In CODE EXAMPLE 2-12, the S1 interface is incorrect, because "+" is predefined for the INTERVAL operand.

CODE EXAMPLE 2-12 User-defined interface that conflicts with the intrinsic use of unary "+".

```
math% cat ce2-12.f95
MODULE M
INTERFACE OPERATOR (+)
    MODULE PROCEDURE S1
END INTERFACE
CONTAINS
REAL FUNCTION S1(X)
    INTERVAL, INTENT(IN) :: X
    S1 = 1.0
END FUNCTION S1
END MODULE M

PROGRAM TEST
USE M
INTERVAL :: X = 0.1
PRINT *, ' + X = ', + X
END PROGRAM TEST

math% f95 -xia ce2-12.f95

MODULE M
       ^
"ce2-12.f95", Line = 1, Column = 8: ERROR: The compiler has
detected errors in module "M".  No module information file will be
created for this module.
```

CODE EXAMPLE 2-12 User-defined interface that conflicts with the intrinsic use of unary "+". *(Continued)*

```
    MODULE PROCEDURE S1
                     ^
"ce2-12.f95", Line = 3, Column = 22: ERROR: This specific interface
"S1" conflicts with the intrinsic use of "+".

USE M
    ^
"ce2-12.f95", Line = 13, Column = 5: ERROR: Module "M" has compile
errors, therefore declarations obtained from the module via the
USE statement may be incomplete.

f90: COMPILE TIME 0.290000 SECONDS
f90: MAXIMUM FIELD LENGTH 4146432 DECIMAL WORDS
f90: 16 SOURCE LINES
f90: 3 ERRORS, 0 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
```

In a generic interface block, if the generic name specified in the `INTERFACE` statement is the name of an intrinsic `INTERVAL` subprogram, the specific user-defined subprograms extend the predefined meaning of the 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.

CODE EXAMPLE 2-13 Correct extension of intrinsic `INTERVAL` function `WID`.

```
math% cat ce2-13.f95
MODULE M
INTERFACE WID
    MODULE PROCEDURE S1
    MODULE PROCEDURE S2
END INTERFACE
CONTAINS
REAL FUNCTION S1(X)
REAL, INTENT(IN) :: X
    S1 = 1.0
END FUNCTION S1
INTERVAL FUNCTION S2(X, Y)
INTERVAL, INTENT(IN) :: X
INTERVAL, INTENT(IN) :: Y
    S2 = [2.0]
END FUNCTION S2
END MODULE M
```

CODE EXAMPLE 2-13 Correct extension of intrinsic INTERVAL function WID. *(Continued)*

```
PROGRAM TEST
USE M
INTERVAL :: X = [1, 2], Y = [3, 4]
REAL     :: R = 0.1
PRINT *, WID(R)
PRINT *, WID(X, Y)

END PROGRAM TEST
math% f95 -xia ce2-13.f95
math% a.out
 1.0
 [2.0,2.0]
```

CODE EXAMPLE 2-14 is correct.

CODE EXAMPLE 2-14 Correct extension of the intrinsic INTERVAL function ABS.

```
math% cat ce2-14.f95
MODULE M
INTERFACE ABS
    MODULE PROCEDURE S1
END INTERFACE
CONTAINS
INTERVAL FUNCTION S1(X)
INTERVAL, INTENT(IN) :: X
    S1 = [-1.0]
END FUNCTION S1
END MODULE M
PROGRAM TEST
USE M
INTERVAL :: X = [1, 2]
PRINT *, ABS(X)

END PROGRAM TEST
math% f95 -xia ce2-14.f95
math% a.out
 [-1.0,-1.0]
```

CODE EXAMPLE 2-15 is correct.

CODE EXAMPLE 2-15 Correct extension of the intrinsic `INTERVAL` function `MIN`.

```
math% cat ce2-15.f95
MODULE M
INTERFACE MIN
    MODULE PROCEDURE S1
END INTERFACE
CONTAINS
INTERVAL FUNCTION S1(X, Y)
    INTERVAL(4), INTENT(IN) :: X
    INTERVAL(8), INTENT(IN) :: Y
    S1 = [-1.0]
END FUNCTION S1
END MODULE M

PROGRAM TEST
USE M
INTERVAL(4) :: X = [1, 2]
INTERVAL(8) :: Y = [3, 4]
REAL        :: R = 0.1
PRINT *, MIN(X, Y)
END PROGRAM TEST
math% f95 -xia ce2-15.f95
math% a.out
 [-1.0,-1.0]
```

2.8.1 Extended Operators with Widest-Need Evaluation

CODE EXAMPLE 2-16 illustrates how widest-need expression processing occurs when calling predefined versus extended versions of an intrinsic INTERVAL operator.

CODE EXAMPLE 2-16 Widest-need expression processing when calling a predefined version of an intrinsic INTERVAL operator.

```
math% cat ce2-16.f95
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 = [1.0]
REAL     :: R = 1.0
COMPLEX  :: C = (1.0, 0.0)
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

math% f95 -xia ce2-16.f95
math% a.out
```

CODE EXAMPLE 2-17 illustrates how widest-need expression processing occurs when calling a user-defined operator.

CODE EXAMPLE 2-17 Widest-need expression processing when invoking a user-defined operator.

```
math% cat ce2-17.f95
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
    PRINT *, ’S1 is invoked.’
    S1 = [1.0]
END FUNCTION S1
INTERVAL FUNCTION S2(X, Y)
INTERVAL, INTENT(IN) :: X
INTERVAL, INTENT(IN) :: Y
    PRINT *, ’S2 is invoked.’
    S2 = [2.0]
END FUNCTION S2
END MODULE M
USE M
INTERVAL :: X = [1.0]
REAL     :: R = 1.0
X = X .AA. R+R     ! S1 is invoked
X = X .AA. X       ! S2 is invoked
END


math% f95 -xia ce2-17.f95

    MODULE PROCEDURE S1
                     ^
"ce2-17.f95", Line = 3, Column = 22: WARNING: Widest-need
evaluation does not apply to arguments of user-defined operation.

USE M
    ^
"ce2-17.f95", Line = 20, Column = 5: WARNING: Widest-need
evaluation does not apply to arguments of user-defined operation.
```

CODE EXAMPLE 2-17 Widest-need expression processing when invoking a user-defined operator. *(Continued)*

```
f90: COMPILE TIME 0.700000 SECONDS
f90: MAXIMUM FIELD LENGTH 5605590 DECIMAL WORDS
f90: 26 SOURCE LINES
f90: 0 ERRORS, 2 WARNINGS, 0 OTHER MESSAGES, 0 ANSI
math% a.out
 S1 is invoked.
 S2 is invoked.
```

2.8.2 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: INTERVAL

If KIND is present, its value is used to determine the result's KTPV; otherwise, the result's KTPV is the same as a default interval.

Containment:

Containment is guaranteed if X is an interval. For example, given

`INTERVAL(16):: X`,

the result of `INTERVAL(X, KIND=4)` contains the INTERVAL X.

However, given `REAL(8) X`, Y, the result of `INTERVAL(X,Y,KIND=4)` does not necessarily contain the internal interval `X .IH. Y`. The reason is that X and Y can be REAL expressions, the values of which cannot be guaranteed.

The INTERVAL constructor does not necessarily contain the value of a literal INTERVAL constant with the same endpoints. For example, `INTERVAL(1.1,1.3)` does not necessarily contain the external value ev(`[1.1,1.3]`) = [1.1, 1.3]. The reason is that the internal values of REAL constants are approximations with unknown accuracy.

To construct an interval that always contains two REAL values, use the interval hull operator .IH., as shown in CODE EXAMPLE 2-18 on page 84.

Result value: The interval result value is a valid interval.

If Y is absent and X is an interval, then INTERVAL(X [,KIND]) is an interval containing X and 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 (and are therefore not intervals), then INTERVAL(X,Y [,KIND]) is an interval with left and right endpoints equal to REAL(X [,KIND]) and REAL(Y [,KIND]) respectively.

Note – In this case, rounding direction is not specified. Therefore, containment is not provided.

[-inf,inf] is returned in two cases:

- If both X and Y are present and Y is less than X.
- If either X or Y or both do not represent a mathematical integer or real number (for example, when one or both REAL arguments is a NaN).

2.8.2.1 Limiting the Scope of Widest-Need

The intrinsic INTERVAL constructor function can be used in two ways:

- To perform KTPV conversions of INTERVAL variables or expressions
- To insulate a non-INTERVAL expression from mixed-mode INTERVAL expression evaluation.

Given the non-INTERVAL (REAL or INTEGER) expression, *EXP*, the code

```
INTERVAL Y
REAL R
R = EXP
Y = R
```

is the same as

```
INTERVAL Y
Y = INTERVAL(EXP)
```

This is not the same as

```
INTERVAL Y
Y = EXP
```

The later will evaluate *EXP* as an interval expression. In the first two code fragments, the expression *EXP* is evaluated as a non-`INTERVAL` expression, and the result is used to construct a degenerate interval.

With two arguments, EXP_1 and EXP_2, `INTERVAL(`EXP_1, EXP_2`)` insulates both expressions from widest-need expression processing and constructs an interval with endpoints equal to the result of the non-`INTERVAL` evaluation of the expressions.

Including the KIND parameter makes it possible to control the KTPV of the result. This is most often needed under -strict expression processing where explicit KTPV conversions are necessary.

The intrinsic `INTERVAL` function with non-`INTERVAL` arguments should be used with care. Whenever interval containment is desired, use the interval hull operator `.IH.`, as shown in CODE EXAMPLE 2-18 on page 84.

The `INTERVAL` constructor acts as a boundary between `INTERVAL` and `REAL` or `INTEGER` expressions. On the non-`INTERVAL` side of this boundary, accuracy (and therefore containment) guarantees cannot be enforced.

CODE EXAMPLE 2-18 Containment Using the `.IH.` Operator

```
math% cat ce2-18.f95
REAL(16) :: A, B
INTERVAL :: X1, X2
PRINT *, "Press Control/D to terminate!"
WRITE(*, 1, ADVANCE='NO')
READ(*, *, IOSTAT=IOS) A, B
DO WHILE (IOS >= 0)
    PRINT *, " FOR A =", A, ", AND B =", B

    ! Widest need code
     X1 = A .IH. B

    ! Equivalent strict code
    X2 = INTERVAL(INTERVAL(A, KIND=16) .IH. INTERVAL(B, KIND=16))
    IF (X1 .SEQ. X2)  PRINT *, 'Check.'
    PRINT *, 'X1 = ', X1
    WRITE(*, 1, ADVANCE='NO')
    READ(*, *, IOSTAT=IOS)  A, B
END DO
1  FORMAT(" A, B = ")
END
math% f95 -xia ce2-18.f95
math% a.out
 Press Control/D to terminate!
 A, B = 1.3 1.7
 FOR A = 1.3 , AND B = 1.7
 X1 =  [1.2999999999999998,1.7000000000000002]
 A, B = 0.0  0.2
 FOR A = 0.0E+0 , AND B = 0.2
 X1 =  [0.0E+0,0.20000000000000002]
 A, B = <Control-D>
```

See Section 2.8.2 "`INTERVAL (X [,Y, KIND])`" on page 81 for details on the use of the intrinsic `INTERVAL` constructor function.

2.8.2.2 KTPV-Specific Names of Intrinsic INTERVAL Constructor Functions

As shown in TABLE 2-11, the intrinsic INTERVAL constructor function can be called using a KTPV-specific form that does not use the optional KIND parameter.

TABLE 2-11 KTPV Specific Forms of the Intrinsic INTERVAL Constructor Function

KTPV -Specific Name	Result
DINTERVAL(X[,Y])	INTERVAL(X[,Y], KIND = 8) or INTERVAL(X[,Y])
SINTERVAL(X[,Y])	INTERVAL(X[,Y], KIND = 4)
QINTERVAL(X[,Y])	INTERVAL(X[,Y], KIND = 16)

2.8.2.3 Intrinsic INTERVAL Constructor Function Conversion Examples

The three examples in this section illustrate how to use the intrinsic INTERVAL constructor to perform conversions from REAL to INTERVAL type data items. CODE EXAMPLE 2-19 shows that REAL expression arguments of the INTERVAL constructor are evaluated using REAL arithmetic and are, therefore, insulated from widest-need expression evaluations.

CODE EXAMPLE 2-19 INTERVAL Conversion

```
math% cat ce2-19.f95
REAL        :: R = 0.1, S = 0.2, T = 0.3
REAL(8)     :: R8 = 0.1D0, T1, T2
INTERVAL(4) :: X, Y
INTERVAL(8) :: DX, DY
R = 0.1
Y  = INTERVAL(R, R, KIND=4)
X  = INTERVAL(0.1, KIND=4)                              ! Line 7
IF ( X == Y ) PRINT *, 'Check1'
X  = INTERVAL(0.1, 0.1, KIND=4)                         ! Line 10
IF ( X == Y ) PRINT *, 'Check2'
T1 = R+S
T2 = T+R8
DY = INTERVAL(T1, T2)
DX = INTERVAL(R+S, T+R8)                                ! Line 15
IF ( DX == DY ) PRINT *, 'Check3'
DX = INTERVAL(Y, KIND=8)                                ! Line 17
IF (Y .CEQ. INTERVAL(0.1, 0.1, KIND=8)) PRINT *, 'Check4'
END
```

CODE EXAMPLE 2-19 INTERVAL Conversion *(Continued)*

```
math% f95 -xia ce2-19.f95
math% a.out
 Check1
 Check2
 Check3
 Check4
```

CODE EXAMPLE 2-19 notes:

- Lines 7 and 10: Interval X is assigned a degenerate interval with both endpoints equal to the internal representation of the real constant 0.1
- Line 15: Interval DX is assigned an interval with left and right endpoints equal to the result of REAL expressions R+S and T+R8 respectively
- Line 17: Interval Y is converted to a containing KTPV-8 interval.

CODE EXAMPLE 2-20 shows how the INTERVAL constructor can be used to construct the smallest possible interval, Y, such that the endpoints of Y are *not* elements of a given interval, X.

CODE EXAMPLE 2-20 Create a narrow interval containing a given real number.

```
math% cat ce2-20.f95
INTERVAL :: X = [10.E-10,11.E+10]
INTERVAL :: Y
Y = INTERVAL(-TINY(INF(X)), TINY(INF(X))) + X
PRINT *, X .INT. Y
END
%math f95 -xia ce2-20
%math a.out
  T
```

Given an interval X, a sharp interval Y satisfying the condition X .INT. Y is constructed. For information on the interior set relation, Section 2.7.3 "Interior: (X .INT. Y)" on page 64.

CODE EXAMPLE 2-21 on page 87 illustrates when the INTERVAL constructor returns the interval [-INF, INF] and [MAX_FLOAT, INF].

CODE EXAMPLE 2-21 `INTERVAL(NaN)`

```
math% cat ce2-21.f95
REAL :: R = 0., S = 0.
T = R/S                        ! Line 2
PRINT *, T
PRINT *, INTERVAL(T, S)        ! Line 4
PRINT *, INTERVAL(T, T)        ! Line 5
PRINT *, INTERVAL(2., 1.)      ! Line 6
PRINT *, INTERVAL(1./R)        ! Line 7
END

math% f95 -xia ce2-21.f95
math% a.out
 NaN
 [-Inf,Inf]
 [-Inf,Inf]
 [-Inf,Inf]
 [1.7976931348623157E+308,Inf]
```

CODE EXAMPLE 2-21 notes:

- Line 2: Variable `T` is assigned a `NaN` value.
- Lines 4 and 5: One of the arguments of the `INTERVAL` constructor is a `NaN` and the result is the interval `[-INF, INF]`.
- Line 6: The interval `[-INF, INF]` is constructed instead of an invalid interval [2,1].
- Line 7: The interval [`MAX_FLOAT`, `INF`] is constructed that contains the interval [`INF`, `INF`]. See the supplementary paper [8] cited in Section 2.10 "References" on page 126, for a discussion of the chosen intervals to internally represent.

2.8.3 Specific Names for Intrinsic Generic `INTERVAL` Functions

The `f95` specific names for intrinsic generic `INTERVAL` functions 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 `f95`, only the following specific intrinsic functions are supported for the `INTERVAL(16)` data type: `VQABS`, `VQAINT`, `VQANINT`, `VQINF`, `VQSUP`, `VQMID`, `VQMAG`, `VQMIG`, and `VQISEMPTY`.

To avoid name space clashes in non-interval programs, the specific names are made available only by the command line options:

- `-xinterval`
- `-xinterval=strict` or `-xinterval=widestneed`
- macro `-xia`, `-xia=strict` or `-xia=widestneed`

See Section 2.3.3 “Interval Command-Line Options” on page 52 for more information.

All the supported intrinsic functions have specific names. For example, TABLE 2-12 lists the names for the `INTERVAL` version of the `ABS` intrinsic.

TABLE 2-12 Specific Names for the Intrinsic `INTERVAL ABS` Function

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

The remaining specific intrinsic functions are listed in Section 2.9.4.4 “Intrinsic Functions” on page 120.

2.9 `INTERVAL` Statements

This section describes the `INTERVAL` statements recognized by `f95`. The syntax and description of each statement is given, along with possible restrictions and examples.

2.9.1 Type Declaration

An `INTERVAL` statement is used to declare `INTERVAL` named constants, variables, and function results. `INTERVAL` is an intrinsic numeric type declaration statement with the same syntax and semantics as standard numeric type declaration statements. The same specifications are available for use with the `INTERVAL` statement as exist for use in other numeric type declarations.

Description: The declaration can be `INTERVAL`, `INTERVAL(4)`, `INTERVAL(8)`, or `INTERVAL(16)`.

2.9.1.1 `INTERVAL`

For a declaration such as

```
INTERVAL :: W
```

the variable, `W`, has the default `INTERVAL` KTPV of 8 and occupies 16 bytes of contiguous memory. In Sun WorkShop 6 Fortran 95, the default `INTERVAL` KTPV is not altered by the command-line options `-xtypemap` or `-r8const`.

`INTERVAL` cannot be used as a derived type name. For example the code in CODE EXAMPLE 2-22 is illegal.

CODE EXAMPLE 2-22 Illegal Derived Type: `INTERVAL`

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

2.9.1.2 `INTERVAL(`n`)`, for $n \in \{4, 8, 16\}$

For a declaration such as

```
INTERVAL(n) :: W
```

the variable, `W`, has KTPV = n and occupies $2n$ bytes of contiguous memory.

CODE EXAMPLE 2-23 on page 90 contains `INTERVAL` variable declarations with different KTPVs. Widest-need and strict value alignment is also shown.

CODE EXAMPLE 2-23 Declaring Intervals with Different KTPVs

```
math% cat ce2-23.f95
INTERVAL(4)  :: X1, X2
INTERVAL(8)  :: Y1, Y2
INTERVAL(16) :: Z1, Z2
REAL(8)      :: D = 1.2345

! Widest-need code
 X1 = D
 Y1 = D
 Z1 = D


! Equivalent strict code
X2 = INTERVAL(INTERVAL(D, KIND=8), KIND=4)
Y2 = INTERVAL(D, KIND=8)
Z2 = INTERVAL(D, KIND=16)

IF (X1 == X2) PRINT *, ’Check1’
IF (Y1 == Y2) PRINT *, ’Check2’
IF (Z1 == Z2) PRINT *, ’Check3’
END

math% f95 -xia ce2-23.f95
math% a.out
 Check1
 Check2
 Check3
```

CODE EXAMPLE 2-24 illustrates how to declare and initialize `INTERVAL` variables. See Section 2.1.2 “`INTERVAL` Constants” on page 42 regarding the different ways to represent `INTERVAL` constants.

CODE EXAMPLE 2-24 Declaring and Initializing INTERVAL Variables

```
math% cat ce2-24.f95
INTERVAL :: U = [1, 9.1_8], V = [4.1]

! Widest-need code
INTERVAL :: W1 = 0.1_16

! Equivalent strict code
INTERVAL :: W2 = [0.1_16]

PRINT *, U, V
IF (W1 .SEQ. W2) PRINT *, 'Check'
END

math% f95 -xia ce2-24.f95
math% a.out
 [1.0,9.1000000000000015] [4.0999999999999996,4.1000000000000006]
 Check
```

In any initializing declaration statement, if the type of the data expression does not match the type of the symbolic name, type conversion is performed.

CODE EXAMPLE 2-25 Declaring INTERVAL Arrays

```
INTERVAL(4) :: R(5), S(5)
INTERVAL :: U(5), V(5)
INTERVAL(16) :: X(5), Y(5)
```

2.9.1.3 DATA Statements

Syntax

The syntax for DATA statements containing INTERVAL variables is the same as for other numeric data types except that INTERVAL variables are initialized using INTERVAL constants.

CODE EXAMPLE 2-26 DATA Statement Containing INTERVAL Variables

```
INTERVAL X
DATA X/[1,2]/
```

2.9.1.4 `EQUIVALENCE` Statements

Any `INTERVAL` variables or arrays may appear in an `EQUIVALENCE` statement with the following restriction: If an equivalence set contains an `INTERVAL` variable or array, all of the objects in the equivalence set must have the same type, as shown in CODE EXAMPLE 1-18 on page 38. This is a standard, not interval-specific, Fortran restriction.

2.9.1.5 `FORMAT` Statements

Syntax

The repeatable edit descriptors for intervals are:

F*w.d,* VF*w.d,* D*w.d*, VD*w.d*, D*w.d*E*e*, VD*w.d*E*e*, Y*w.d*, and Y*w.d*E*e*

where

D ∈ {E, EN, ES, G}

w and *e* are nonzero, unsigned integer constants

d is an unsigned integer constant.

See Section 2.9.2 “Input and Output” on page 98 for the specifications of how edit descriptors process `INTERVAL` data. For the behavior of standard edit descriptors with non-`INTERVAL` data, see the *Fortran Reference Manual*.

All standard Fortran edit descriptors accept intervals. The prefix `V` can be added to the standard `E`, `F`, and `G` edit descriptors for interval-only versions.

As shown in CODE EXAMPLE 2-27, no modifications to nonrepeatable edit descriptors are required when reading or writing `INTERVAL` data.

CODE EXAMPLE 2-27 Nonrepeatable Edit Descriptor Example

```
math% cat ce2-27.f95
INTERVAL :: X = [-1.3, 1.3]
WRITE(*, '(SP, VF20.5)') X
WRITE(*, '(SS, VF20.5)') X
END
math% f95 -xia ce2-27.f95
math% a.out
 [-1.30001,+1.30001]
 [-1.30001, 1.30001]
```

Description

Repeatable Edit Descriptors

The repeatable edit descriptors `E`, `F`, `EN`, `ES`, `G`, `VE`, `VEN`, `VES`, `VF`, `VG`, and `Y` specify how `INTERVAL` data are edited.

CODE EXAMPLE 2-28 contains examples of `INTERVAL`-specific edit descriptors.

CODE EXAMPLE 2-28 Format Statements with `INTERVAL`-specific Edit Descriptors

```
FORMAT(VE22.4E4)
FORMAT(VEN22.4)
FORMAT(VES25.5)
FORMAT(VF25.5)
FORMAT(VG25.5)
FORMAT(VG22.4E4)
FORMAT(Y25.5)
```

See Section 2.9.2 "Input and Output" on page 98 for additional examples.

2.9.1.6 `FUNCTION` (External)

As shown in CODE EXAMPLE 2-29, there is no difference between an interval and a non-interval external function, except for the use of `INTERVAL` types (`INTERVAL`, `INTERVAL(4)`, `INTERVAL(8)`, or `INTERVAL(16)`) in the function's and argument's definitions.

CODE EXAMPLE 2-29 Default Interval Function

```
INTERVAL FUNCTION SQR (A)                    ! Line 1
INTERVAL :: A
SQR = A**2
RETURN
END
```

The default `INTERVAL` in line 1 can be made explicit, as shown in CODE EXAMPLE 2-30.

CODE EXAMPLE 2-30 Explicit `INTERVAL(16)` Function Declaration

```
INTERVAL(16) FUNCTION SQR (A)                ! Line 1
```

2.9.1.7 `IMPLICIT` Attribute

Use the `IMPLICIT` attribute to specify the default type of interval names.

```
IMPLICIT INTERVAL (8) (V)
```

2.9.1.8 `INTRINSIC` Statement

Use the `INTRINSIC` statement to declare intrinsic functions, so they can be passed as actual arguments.

CODE EXAMPLE 2-31 Intrinsic Function Declaration

```
INTRINSIC VDSIN, VDCOS, VQSIN
X = CALC(VDSIN, VDCOS, VQSIN)
```

Note – Specific names of generic functions must be used in the `INTRINSIC` statement and passed as actual arguments. See Section 2.8.3 "Specific Names for Intrinsic Generic `INTERVAL` Functions" on page 87 and Section 2.9.4.4 "Intrinsic Functions" on page 120.

Because they are generic, the following intrinsic `INTERVAL` functions cannot be passed as actual arguments:

```
NDIGITS, INTERVAL
```

2.9.1.9 `NAMELIST` Statement

The `NAMELIST` statement supports intervals.

CODE EXAMPLE 2-32 `INTERVALS` in a `NAMELIST`

```
CHARACTER(8) :: NAME
CHARACTER(4) :: COLOR
INTEGER      :: AGE
INTERVAL(4)  :: HEIGHT
INTERVAL(4)  :: WEIGHT
NAMELIST /DOG/ NAME, COLOR, AGE, WEIGHT, HEIGHT
```

2.9.1.10 `PARAMETER` Attribute

The `PARAMETER` attribute is used to assign the result of an `INTERVAL` initialization to a named constant (`PARAMETER`).

Syntax

`PARAMETER` (*p* = *e* [, *p* = *`expr`*]...)

p `INTERVAL` symbolic name

`expr` `INTERVAL` constant expression

= assigns the value of *e* to the symbolic name, *p*

Description

Both the symbolic name, *p*, and the constant expression, *`expr`*, must have `INTERVAL` types.

Exponentiation to an integer power is allowed in constant expressions.

Mixed-mode `INTERVAL` expression evaluation is supported in the definition of interval named constants under widest-need expression processing. If the constant expression's type does not match the named constant's type, type conversion of the constant expression is performed under widest-need expression processing.

Caution – In `f95`, non-`INTERVAL` constant expressions are evaluated at compile time without regard to their possible subsequent use in mixed-mode `INTERVAL` expressions. They are outside the scope of widest-need expression processing. Therefore, no requirement exists to contain the value of the point expression used to set the value of non-`INTERVAL` named constants. To remind users whenever a non-`INTERVAL` named constant appears in a mixed-mode `INTERVAL` expression, a compile-time warning message is issued. Named constants, as defined by the Fortran standard, are more properly called *read-only variables*. There is no external value associated with a read-only variable.

In standard Fortran 95, named constants cannot be used to represent the infimum and supremum of an `INTERVAL` constant. This is a known error that this constraint is not enforced in this release.

CODE EXAMPLE 2-33 Constant Expression in Non-INTERVAL PARAMETER Attribute

```
math% cat ce2-33.f95
REAL(4), PARAMETER      :: R   = 0.1
INTERVAL(4), PARAMETER  :: I4  = 0.1
INTERVAL(16), PARAMETER :: I16 = 0.1
INTERVAL                :: XR, XI
XR = R4
XI = I4
IF ((.NOT.(XR.SP.I16)).AND. (XI.SP.I16)) PRINT *, 'Check.'
END
math% f95 -xia ce2-33.f95
math% a.out
 Check.
```

Note – XR does not contain 1/10, whereas XI does.

2.9.1.11 Fortran 95-Style POINTER

Intervals can be associated with pointers.

CODE EXAMPLE 2-34 INTERVAL Pointers

```
INTERVAL, POINTER :: PX
INTERVAL :: X
X => P
```

2.9.1.12 Statement Function

A statement function can be used to declare and evaluate parameterized INTERVAL expressions. Non-INTERVAL statement function restrictions apply.

CODE EXAMPLE 2-35 INTERVAL Statement Function

```
math% cat ce2-35.f95
INTERVAL :: X, F
F(X) = SIN(X)**2 + COS(X)**2
IF(1 .IN. F([0.5])) PRINT *, 'Check'
END
math% f95 -xia ce2-35.f95
math% a.out
 Check
```

2.9.1.13 Type Statement

The type statement specifies the data type of variables in a variable list. Optionally the type statement specifies array dimensions, and initializes variables with values.

Syntax

The syntax is the same as for non-INTERVAL numeric data types, except that type can be one of the following INTERVAL type specifiers: INTERVAL, INTERVAL(4), INTERVAL(8), or INTERVAL(16).

Description

Properties of the type statement are the same for INTERVAL types as for other numeric data types.

Restrictions

Same as for non-INTERVAL numeric types.

CODE EXAMPLE 2-36 INTERVAL Type Statement

```
INTERVAL     :: I, J = [0.0]
INTERVAL(16) :: K = [0.1, 0.2_16]
INTERVAL(16) :: L = [0.1]
```

CODE EXAMPLE 2-36 notes:
- J is initialized to [0.0]
- K is initialized to an interval containing [0.1, 0.2]
- L is initialized to an interval containing [0.1]

2.9.1.14 `WRITE` Statement

The `WRITE` statement accepts `INTERVAL` variables and processes an input/output list in the same way that non-`INTERVAL` type variables are processed. Formatted writing of `INTERVAL` data is performed using the defined `INTERVAL` edit descriptors. `NAMELIST`-directed `WRITE` statements support intervals.

2.9.1.15 `READ` Statement

The `READ` statement accepts `INTERVAL` variables and processes an input/output list in the same way that non-`INTERVAL` type variables are processed.

2.9.2 Input and Output

The process of performing `INTERVAL` input/output is the same as for other non-`INTERVAL` data types.

2.9.2.1 External Representations

Let x be an external (decimal) number that can be read or written using either list-directed or formatted input/output. See the subsections in Section 2.1 "Fortran Extensions" regarding the regarding the distinction between internal approximations and external values. Such a number can be used to represent either an external interval, or an endpoint. There are three displayable forms of an external interval:

- `[X_inf, X_sup]` represents the mathematical interval $[\underline{x}, \bar{x}]$
- `[X]` represents the degenerate mathematical interval $[x, x]$, or $[x]$
- `X` represents the non-degenerate mathematical interval $[x] + [-1,+1]_{\text{uld}}$ (unit in the last digit). This form is the single-number representation, in which the last decimal digit is used to construct an interval (see the `Y` edit descriptor). In this form, trailing zeros are significant. Thus `0.10` represents interval `[0.09, 0.11]`, `100E-1` represents interval `[9.9, 10.1]`, and `0.10000000` represents the interval `[0.099999999, 0.100000001]`.

A positive or negative infinite interval endpoint is input/output as a case-insensitive string `INF` or `INFINITY` prefixed with a minus or an optional plus sign.

The empty interval is input/output as the case-insensitive string `EMPTY` enclosed in square brackets, `"[...]"`. The string, `EMPTY,` may be preceded or followed by blanks.

CODE EXAMPLE 1-6 on page 22 can be used to experiment with extended intervals.

See Section 2.4.1 "Arithmetic Operators `+`, `–`, `*`, `/`" on page 56, for more details.

2.9.2.2 Input

On input, any external interval, `X`, or its components, `X_inf` and `X_sup`, can be formatted in any way that is accepted by the D*w.d* edit descriptor. Therefore, let *input-field*, *input-field*$_1$, and *input-field*$_2$ be valid input fields for the Dw'.*d*, Dw_1.*d*, and Dw_2.*d* edit descriptors, respectively.

Let *w* be the width of an interval input field. On input, *w* must be greater than zero. All `INTERVAL` edit descriptors accept input `INTERVAL` data in each of the following three forms:

- [*input-field1*, *input-field2*], in which case $w_1 + w_2 = w - 3$ or $w = w_1 + w_2 + 3$
- [*input-field*], in which case $w' = w$-2 or $w = w'$+2
- *input-field*, in which case $w' = w$

The first form (two numbers enclosed in brackets and separated by a comma) is the familiar [*inf*, *sup*] representation.

The second form (a single number enclosed in brackets) denotes a point or degenerate interval.

The third form (without brackets) is the single-number form of an interval in which the last displayed digit is used to determine the interval's width. See Section 2.9.2.7 "Single-Number Editing with the Y Edit Descriptor" on page 105. For more detailed information, see M. Schulte, V. Zelov, G.W. Walster, D. Chiriaev, "Single-Number Interval I/O," *Developments in Reliable Computing*, T. Csendes (ed.), (Kluwer 1999).

If an infimum is not internally representable, it is rounded down to an internal approximation known to be less than the exact value. If a supremum is not internally representable, it is rounded up to an internal approximations known to be greater than the exact input value. If the degenerate interval is not internally representable, it is rounded down and rounded up to form an internal `INTERVAL` approximation known to contain the exact input value.

2.9.2.3 List-Directed Input

If an input list item is an `INTERVAL`, the corresponding element in the input record must be an external interval or a null value.

An external 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 with no enclosing square brackets, "["... "]", the external interval is interpreted using the single-number interval representation: $[x] + [-1,1]_{\text{uld}}$ (unit in the last digit).

When using the [*inf*, *sup*] input style, an end of record may occur between the infimum and the comma or between the comma and the supremum.

A null value, specified by two consecutive commas, means that the corresponding INTERVAL list item is unchanged.

Note – Do not use a null value for the infimum or supremum of an interval.

CODE EXAMPLE 2-37 List Directed Input/Output Code

```
math% cat ce2-37.f95
INTERVAL, DIMENSION(6) :: X
INTEGER I
DO I = LBOUND(X, 1), UBOUND(X, 1)
    READ(*, *) X(I)
    WRITE(*, *) X(I)
END DO
END
math% f95 -xia ce2-37.f95
math% a.out
1.234500
 [1.2344989999999997,1.2345010000000001]
[1.2345]
 [1.2344999999999999,1.2345000000000002]
[-inf,2]
 [-Inf,2.0]
[-inf]
 [-Inf,-1.7976931348623157E+308]
[EMPTY]
 [EMPTY]
[1.2345,1.23456]
 [1.2344999999999999,1.2345600000000002]
```

2.9.2.4 Formatted Input/Output

The INTERVAL edit descriptors are:

- E*w.d*E*e*
- EN*w.d*
- ES*w.d*
- F*w.d*
- G*w.d*E*e*
- VE*w.d*E*e*
- VEN*w.d*E*e*
- VES*w.d*E*e*
- VF*w.d*
- VG*w.d*E
- Y*w.d*E*e*

In the `INTERVAL` edit descriptors:

- *w* specifies the number of positions occupied by the field
- *d* specifies the number of digits to the right of the decimal point
- E*e* specifies the width of exponent field

The parameters *w* and *d* must be used. E*e* is optional

The *w* and *d* specifiers must be present and are subject to the following constraints:

- $e > 0$
- $w \geq 0$ when using the `F` edit descriptor, or $w > 0$ when using all edit descriptors other than `F`.

Input Actions

Input actions for formatted interval input are the same as for other numeric data types, except that in all cases, the stored internal approximation contains the external value represented by the input character string. Containment can require outward rounding of interval endpoints. Given any input interval characters, `input_string`, the corresponding external value, ev(*`input_string`*), and the resulting internal approximation after input conversion, `X`, are related:

ev(*`input_string`*) $\subseteq$ `X`.

During input, all interval edit descriptors have the same semantics. The value of parameter *w*, is the field width containing the external interval. The value of *e* is ignored.

Output Actions

Output actions for formatted interval output are the same as for other data types, except that in all cases, the mathematical value of the output character string must contain the mathematical value of the internal data item in the output list. Containment can require outward rounding of interval endpoints. Given any internal interval, `X`, the corresponding output characters, `output_string`, and the external value, ev(*`output_string`*), are related:

`X` $\subseteq$ ev(*`output_string`*).

During output, edit descriptors cause the internal value of interval output list items to be displayed using different formats. However, the containment constraint requires that

ev(*`input_string`*) $\subseteq$ `X` $\subseteq$ ev(*`output_string`*)

2.9.2.5 Formatted Input

The behavior of formatted input is identical for all INTERVAL edit descriptors listed in Section 2.9.2.4 "Formatted Input/Output" on page 100. All inputs described in Section 2.9.2.2 "Input" on page 99 are accepted.

If the input field contains a decimal point, the value of *d* is ignored. If a decimal point is omitted from the input field, *d* determines the position of the decimal point in the input value; that is, the input value is read as an integer and multiplied by $10^{(-d)}$.

CODE EXAMPLE 2-38 The decimal point in an input value dominates format specifiers.

```
math% cat ce2-38.f95
INTERVAL :: X, Y
READ(*, '(F10.4)') X
READ(*, '(F10.4)') Y
WRITE(*, *)'1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, E19.6)') X
WRITE(*, '(1X, E19.6)') Y
END
math% f95 -xia ce2-38.f95
math% a.out
[.1234]
[1234]
 1234567890123456789012345678901234567890-position
       0.123400E+000
       0.123400E+000
```

CODE EXAMPLE 2-39 All the INTERVAL edit descriptors accept single-number input.

```
math% cat ce2-39.f95
INTERVAL, DIMENSION(9) :: X
INTEGER                :: I
READ(*, '(Y25.3)')   X(1)
READ(*, '(E25.3)')   X(2)
READ(*, '(F25.3)')   X(3)
READ(*, '(G25.3) ')  X(4)
READ(*, '(VE25.3)')  X(5)
READ(*, '(VEN25.3)') X(6)
READ(*, '(VES25.3)') X(7)
READ(*, '(VF25.3)')  X(8)
READ(*, '(VG25.3)')  X(9)
DO I = LBOUND(X, 1), UBOUND(X, 1)
    PRINT *, X(I)
END DO
END
%math f95 -xia ce2-39.f95
%math a.out
1.23
1.23
1.23
1.23
1.23
1.23
1.23
1.23
1.23
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
 [1.2199999999999999,1.2400000000000003]
```

Blank Editing (BZ)

Because trailing zeros are significant in single-number INTERVAL input, the BZ control edit descriptor is ignored when processing leading and trailing blanks for input to INTERVAL list items.

CODE EXAMPLE 2-40 BZ Descriptor

```
math% cat ce2-40.f95
INTERVAL :: X
REAL(4)  :: R
READ(*, '(BZ, F40.6 )') X
READ(*, '(BZ, F40.6 )') R
WRITE(*, '(VF40.3)')    X
WRITE(*, '(F40.3)')     R
END
math% f95 -xia ce2-40.f95
math% a.out
[.9998   ]
    .9998
 [             0.999,             1.000]
                                   1.000
```

Scale Factor (P)

The P edit descriptor changes the scale factor for Y, VE, VEN, VES, VF, and VG descriptors and for F, E, EN, ES, and G edit descriptors when applied to intervals. The P edit descriptor scales interval endpoints the same way it scales REAL values.

2.9.2.6 Formatted Output

The F, E, and G edit descriptors applied to intervals have the same meaning as the Y edit descriptor except that if the F or G edit descriptor is used, the output field may be formatted using the F edit descriptor. If the E edit descriptor is used, the output field always has the form prescribed by the E edit descriptor.

Formatted INTERVAL output has the following properties:

- A positive interval endpoint starts with an optional plus sign.
- A negative endpoint always starts with a leading minus sign.
- A zero interval endpoint never starts with a leading plus or minus.
- The VF, VE and VG edit descriptors provide [*inf*, *sup*]-style formatting of intervals.
- The Y edit descriptor produces single-number interval output.

- If an output list item matching the `VF`, `VE`, `VG`, or `Y` edit descriptor is any type other than `INTERVAL`, the entire output field is filled with asterisks.
- If the output field's width, *w*, in `VF`, `VE`, or `VG` edit descriptors is an even number, the field is filled with one leading blank character and *w*-1 is used for the output field's width.

On output, the default values for the exponent field, *e*, are shown in TABLE 2-13.

TABLE 2-13 Default Values for Exponent Field in Output Edit Descriptors

Edit Descriptor	`INTERVAL(4)`	`INTERVAL(8)`	`INTERVAL(16)`
`Y, E, EN, ES, G`	3	3	3
`VE, VEN, VES, VG`	2	2	3

2.9.2.7 Single-Number Editing with the `Y` Edit Descriptor

The `Y` edit descriptor formats extended interval values in the single-number form.

If the external `INTERVAL` value is not degenerate, the output format is the same as for a `REAL` or `INTEGER` literal constant (`X` without square brackets, "`[`"..."`]`"). The external value is interpreted as a non-degenerate mathematical interval $[x] + [-1,1]_{uld}$. 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. However, the actual number of displayed digits may be more or less than *d*, depending on the value of *w* and the width of the external interval.

The *e* specifier (if present) defines the number of places in the output subfield reserved for the exponent.

The presence of the *e* specifier forces the output field to have the form prescribed by the `E` (as opposed to `F`) edit descriptor.

The single-number interval representation is often less precise than the [*inf*, *sup*] representation. This is particularly true when an interval or its single-number representation contains zero or infinity.

For example, the external value of the single-number representation for [-15, +75] is ev(`[0E2]`) = [-100, +100]. The external value of the single-number representation for [1, ∞] is ev(`[0E+inf]`) = [-∞, +∞].

In these cases, to produce a narrower external representation of the internal approximation, the `VG`*w*.*d*′`E`*e* edit descriptor is used, with *d*′ ≥ 1 to display the maximum possible number of significant digits within the *w*-character input field.

CODE EXAMPLE 2-41 Y [*inf*, *sup*]-style Output

```
math% cat ce2-41.f95
INTERVAL :: X = [-1, 10]
INTERVAL :: Y = [1, 6]
WRITE(*, '(Y20.5)') X
WRITE(*, '(Y20.5)') Y
END
math% f95 -xia ce2-41.f95
math% a.out
 [-1.     ,0.1E+002]
 [1.0     ,6.0     ]
```

If it is possible to represent a degenerate interval within the *w*-character output field, the output string for a single number is enclosed in obligatory square brackets, "[", ... "]" to signify that the result is a point.

If there is sufficient field width, the E or F edit descriptor is used, depending on which can display the greater number of significant digits. If the number of displayed digits using the E and F edit descriptor is the same, the F edit descriptor is used.

CODE EXAMPLE 2-42 Y*w.d* Output

```
cat math% cat ce2-42.f95
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1x, F20.6)') [1.2345678, 1.23456789]
WRITE(*, '(1x, F20.6)') [1.234567, 1.2345678]
WRITE(*, '(1x, F20.6)') [1.23456, 1.234567]
WRITE(*, '(1x, F20.6)') [1.2345, 1.23456]
WRITE(*, '(1x, F20.6)') [1.5111, 1.5112]
WRITE(*, '(1x, F20.6)') [1.511, 1.512]
WRITE(*, '(1x, F20.6)') [1.51, 1.52]
WRITE(*, '(1x, F20.6)') [1.5, 1.5]
END
math% f95 -xia ce2-42.f95
math% a.out
 1234567890123456789012345678901234567890-position
       1.2345679
       1.234567
       1.23456
       1.2345
       1.511
       1.51
       1.5
 [     1.50000000000]
```

Increasing interval width decreases the number of digits displayed in the single-number representation. When the interval is degenerate all remaining positions are filled with zeros and brackets are added if the degenerate interval value is represented exactly.

The intrinsic function `NDIGITS` (see TABLE 2-21 on page 125) returns the maximum number of significant digits necessary to write an `INTERVAL` variable or array using the single-number display format.

CODE EXAMPLE 2-43 Y*w.d* Output Using the `NDIGITS` Intrinsic

```
math% cat ce2-43.f95
INTEGER :: I, ND, T, D, DIM
PARAMETER(D=5)      ! Some default number of digits
PARAMETER(DIM=8)
INTERVAL, DIMENSION(DIM) :: X
CHARACTER(20) :: FMT
X = (/ [1.2345678, 1.23456789], &
  [1.234567, 1.2345678], &
  [1.23456, 1.234567], &
  [1.2345, 1.23456], &
  [1.5111, 1.5112], &
  [1.511, 1.512], &
  [1.51, 1.52], &
  [1.5]/)
ND=0
DO I=1, DIM
    T = NDIGITS(X(I))
    IF(T == EPHUGE(T)) THEN ! The interval is degenerate
        ND = MAX(ND, D)
    ELSE
        ND = MAX( ND, T )
    ENDIF
END DO

WRITE(FMT, '(A2, I2, A1, I1, A1)') '(E', 10+ND, '.', ND, ')'

DO I=1, DIM
    WRITE(*, FMT) X(I)
END DO
END
```

CODE EXAMPLE 2-43 Y*w.d* Output Using the NDIGITS Intrinsic

```
math% f95 -xia ce2-43.f95
math% a.out
  0.12345679E+001
  0.1234567 E+001
  0.123456  E+001
  0.12345   E+001
  0.1511    E+001
  0.151     E+001
  0.15      E+001
[ 0.15000000E+001]
```

For readability, the decimal point is always located in position $p = e + d + 4$, counting from the right of the output field.

CODE EXAMPLE 2-44 {Y, F, E, G}*w.d* output, where *d* sets the minimum number of significant digits to be displayed.

```
math% cat ce2-44.f95
INTERVAL :: X = [1.2345678, 1.23456789]
INTERVAL :: Y = [1.5]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, F20.5)') X
WRITE(*, '(1X, F20.5)') Y
WRITE(*, '(1X, 1E20.5)') X
WRITE(*, '(1X, 1E20.5)') Y
WRITE(*, '(1X, G20.5)') X
WRITE(*, '(1X, G20.5)') Y
WRITE(*, '(1X, Y20.5)') X
WRITE(*, '(1X, Y20.5)') Y
END
math% f95 -xia ce2-44.f95
math% a.out
 1234567890123456789012345678901234567890-position
        1.2345679
 [      1.5000000000]
        0.12345E+001
 [      0.15000E+001]
        1.2345679
 [      1.5000000000]
        1.2345679
 [      1.5000000000]
```

The optional *e* specifier specifies the number of exponent digits. If the number of exponent digits is specified, *w* must be at least $d + e + 7$.

CODE EXAMPLE 2-45 Y*w*.*d*E*e* output (the usage of *e* specifier).

```
math% cat ce2-45.f95
INTERVAL :: X = [1.2345, 1.2346]
INTERVAL :: Y = [3.4567, 3.4568]
INTERVAL :: Z = [1.5]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, Y19.5E4)') X
WRITE(*, '(1X, Y19.5E4)') Y
WRITE(*, '(1X, Y19.5E4)') Z
WRITE(*, '(1X, Y19.5E3)') X
WRITE(*, '(1X, Y19.5E3)') Y
WRITE(*, '(1X, Y19.5E3)') Z
END
math% f95 -xia ce2-45.f95
math% a.out
 1234567890123456789012345678901234567890-position
      0.1234 E+0001
      0.3456 E+0001
 [    0.15000E+0001]
       0.1234 E+001
       0.3456 E+001
 [     0.15000E+001]
```

2.9.2.8 E Edit Descriptor

The E edit descriptor formats INTERVAL data items using the single-number E form of the Y edit descriptor.

The general form is:

E*w*.*d*E*e*

CODE EXAMPLE 2-46 E*w*.*d*E*e* Edit Descriptor

```
math% cat ce2-46.f95
INTERVAL :: X = [1.2345678, 1.23456789]
INTERVAL :: Y = [1.5]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, E20.5)')   X
WRITE(*, '(1X, E20.5E3)') X
WRITE(*, '(1X, E20.5E3)') Y
WRITE(*, '(1X, E20.5E4)') X
WRITE(*, '(1X, E20.5E2)') X
END
```

CODE EXAMPLE 2-46 E*w.d*E*e* Edit Descriptor *(Continued)*

```
math% f95 -xia ce2-46.f95
math% a.out
 1234567890123456789012345678901234567890-position
        0.12345E+001
        0.12345E+001
 [      0.15000E+001]
       0.12345E+0001
         0.12345E+01
```

2.9.2.9 F Edit Descriptor

The `F` edit descriptor formats `INTERVAL` data items using only the `F` form of the `INTERVAL` `Y` edit descriptor. The general form is:

F*w.d*

Using the `F` descriptor, it is possible to display more significant digits than specified by *d*. Positions corresponding to the digits that are not displayed are filled with blanks.

CODE EXAMPLE 2-47 F*w.d* Edit Descriptor

```
math% cat ce2-47.f95
INTERVAL :: X = [1.2345678, 1.23456789]
INTERVAL :: Y = [2.0]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, F20.4)') X
WRITE(*, '(1X, E20.4)') X
WRITE(*, '(1X, F20.4)') Y
WRITE(*, '(1X, E20.4)') Y
END
math% f95 -xia ce2-47.f95
math% a.out
 1234567890123456789012345678901234567890-position
         1.2345679
         0.1234E+001
 [       2.000000000]
 [       0.2000E+001]
```

2.9.2.10 G Edit Descriptor

The G edit descriptor formats INTERVAL data items using the single-number E or F form of the Y edit descriptor. The general form is:

G*w.d*E*e*

CODE EXAMPLE 2-48 G*w.d*E*e* Edit Descriptor

```
math% cat ce2-48.f95
INTERVAL :: X = [1.2345678, 1.23456789]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, G20.4)')   X
WRITE(*, '(1X, G20.4E3)') X
END
math% f95 -xia ce2-48.f95
math% a.out
 1234567890123456789012345678901234567890-position
         1.2345679
         0.1234E+001
```

Note – If it is impossible to output interval endpoints according to the F descriptor, G edit descriptor uses the E descriptor

2.9.2.11 VE Edit Descriptor

The general form of the VE edit descriptor is:

VE*w.d*E*e*

Let Xd be a valid external value using the E*w′.d* edit descriptor. The VE edit descriptor outputs INTERVAL data items in the following form:

[X_inf,X_sup], where $w' = (w\text{-}3)/2$.

The external values, X_inf and X_sup, are lower and upper bounds, respectively, on the infimum and supremum of the INTERVAL output list item.

CODE EXAMPLE 2-49 VE Output

```
math% cat ce2-49.f95
INTERVAL :: X = [1.2345Q45, 1.2346Q45]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, VE25.3)')   X
WRITE(*, '(1X, VE33.4E4)') X
END
math% f95 -xia ce2-49.f95
math% a.out
 1234567890123456789012345678901234567890-position
 [ 0.123E+046, 0.124E+046]
 [   0.1234E+0046,   0.1235E+0046]
```

2.9.2.12 VEN Edit Descriptor

The general form of the VEN edit descriptor is:

VEN*w*.*d*E*e*

Let X_inf and X_sup be valid external values displayed using the EN*w′*.*d* edit descriptor. The VEN edit descriptor outputs an INTERVAL data item in the following form:

[X_inf,X_sup], where $w' = (w-3)/2$.

The external values, X_inf and X_sup, are lower and upper bounds, respectively, on the infimum and supremum of the INTERVAL output list item.

CODE EXAMPLE 2-50 VEN Output

```
math% cat ce2-50.f95
INTERVAL :: X = [1024.82]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, VEN25.3)') X
WRITE(*, '(1X, VEN33.4E4)') X
END
math% f95 -xia ce2-50.f95
math% a.out
 1234567890123456789012345678901234567890-position
 [ 1.024E+003, 1.025E+003]
 [   1.0248E+0003,   1.0249E+0003]
```

2.9.2.13 VES Edit Descriptor

The general form of the VES edit descriptor is:

VES*w*.*d*E*e*

Let X_inf and X_sup be a valid external values using the ES*w'*.*d* edit descriptor. The VES edit descriptor outputs an INTERVAL data item in the following form:

[X_inf,X_sup], where $w' = (w\text{-}3)/2$.

The external values, X_inf and X_sup, are lower and upper bounds, respectively, on the infimum and supremum of the INTERVAL output list item.

CODE EXAMPLE 2-51 VES Output

```
math% cat ce2-51.f95
INTERVAL :: X = [21.234]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, VES25.3)')   X
WRITE(*, '(1X, VES33.4E4)') X
END
math% f95 -xia ce2-51.f95
math% a.out
 1234567890123456789012345678901234567890-position
 [ 2.123E+001, 2.124E+001]
 [   2.1233E+0001,   2.1235E+0001]
```

2.9.2.14 VF Edit Descriptor

Let X_inf and X_sup be valid external values displayed using the F*w'*.*d* edit descriptor. The VF edit descriptor outputs INTERVAL data items in the following form:

[X_inf,X_sup], where $w' = (w\text{-}3)/2$.

The external values, X_inf and X_sup, are lower and upper bounds, respectively, on the infimum and supremum of the INTERVAL output list item.

CODE EXAMPLE 2-52 VF Output Editing

```
math% cat ce2-52.f95
INTERVAL :: X = [1.2345, 1.2346], Y = [1.2345E11, 1.2346E11]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, VF25.3)') X
WRITE(*, '(1X, VF25.3)') Y
END
math% f95 -xia ce2-52.f95
math% a.out
 1234567890123456789012345678901234567890-position
 [      1.234,      1.235]
 [***********,***********]
```

Note – If it is impossible to output an interval according to the specified format statement, asterisks are printed.

2.9.2.15 VG Edit Descriptor

For INTERVAL output, VG editing is the same as VE or VF editing, except that the G edit descriptor is used to format the displayed interval endpoints.

CODE EXAMPLE 2-53 VG Output

```
math% cat ce2-53.f95
INTERVAL :: X = [1.2345, 1.2346], Y = [1.2345E11, 1.2346E11]
WRITE(*, *) '1234567890123456789012345678901234567890-position'
WRITE(*, '(1X, VG25.3)') X
WRITE(*, '(1X, VG25.3)') Y
END
math% f95 -xia ce2-53.f95
math% a.out
 1234567890123456789012345678901234567890-position
 [  1.23     ,  1.24     ]
 [ 0.123E+012, 0.124E+012]
```

Note – If it is impossible to output interval endpoints according to the F descriptor, the VG edit descriptor uses the E descriptor.

2.9.2.16 Unformatted Input/Output

Unformatted input/output is used to transfer data to and from memory locations without changing its internal representation. With intervals, unformatted input/output is particularly important, because outward rounding on input and output is avoided.

Note – Use only unformatted INTERVAL input and output to read and write unformatted INTERVAL data. Binary file compatibility with future releases is not guaranteed. Unformatted input/output relies on the fact that INTERVAL data items are opaque.

2.9.2.17 List-Directed Output

REAL constants for left and right endpoints are produced using either an F or an E edit descriptor. Let $|x|$ be the absolute value of an output interval endpoint. Then if

$$10^{d_1} \le |x| \le 10^{d_2},$$

the endpoint is produced using the 0PF*w.d* edit descriptor. Otherwise, the 1PE*w.d*E*e* descriptor is used. In f95, d_1 = -2 and d_2 = +8.

For the output of INTERVAL data items in f95, the values for *d* and *e* are the same as for the REAL types with the same KTPV. The value of *w* reflects the need to conveniently accommodate two REAL values and three additional characters for square brackets, "[", "]", and the comma, as shown in CODE EXAMPLE 2-37 on page 100.

2.9.2.18 Single-Number Input/Output and Base Conversions

Single-number INTERVAL 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 increase in the width of the stored input interval. For example, an input of 1.37 followed by an immediate print will result in 1.3 being output. See Section 2.9.2.4 "Formatted Input/Output" on page 100.

As shown in CODE EXAMPLE 1-6 on page 22, programs must use character input and output to exactly echo input values and internal reads to convert input character strings into valid internal approximations.

2.9.3 Intrinsic `INTERVAL` Functions

This section contains the defining properties of the `f95` intrinsic `INTERVAL` functions.

Generic intrinsic `INTERVAL` functions that accept arguments with more than one KTPV have both generic and KTPV-specific names. When an intrinsic function is invoked using its KTPV-specific name, arguments must have the matching KTPV.

Note – In `f95`, some KTPV-16 specific intrinsic functions are not provided. This is an outstanding quality of implementation opportunity.

With functions that accept more than one `INTERVAL` data item (for example, `SIGN(A,B)`), all arguments must have the same KTPV. Under widest-need expression processing, compliance with this restriction is automatic. With strict expression processing, developers are responsible for enforcing type and KTPV restrictions on intrinsic function arguments. Compile-time errors result when different KTPVs are encountered.

2.9.4 Mathematical Functions

This section lists the type-conversion, trigonometric, and other functions that accept `INTERVAL` arguments. The symbols $\underline{x}$ and $\bar{x}$ in the interval $[\underline{x}, \bar{x}]$ are used to denote its ordered elements, the infimum, or lower bound and supremum, or upper bound, respectively. In point (non-interval) function definitions, lowercase letters *x* and *y* are used to denote `REAL` or `INTEGER` values.

When evaluating a function, *f*, of an interval argument, *X*, the interval result, *f*(*X*), must be an enclosure of its containment set, cset(*f*, {*x*}), where:

$$\text{cset}(f, \{X\}) = \{\text{cset}(f, \{x\}) | \ x \in X\}$$

A similar result holds for functions of *n*-variables. Determining the containment set of values that must be included when the interval $[\underline{x}, \bar{x}]$ contains values outside the domain of *f* is discussed in the supplementary paper [1] cited in Section 2.10 "References" on page 126. The results therein are needed to determine the set of values that a function can produce when evaluated on the boundary of, or outside its domain of definition. This set of values, called the *containment set* is the key to defining interval systems that return valid results, no matter what the value of a function's arguments or an operator's operands. As a consequence, there are no argument restrictions on any intrinsic `INTERVAL` functions in `f95`.

2.9.4.1 Inverse Tangent Function `ATAN2(Y,X)`

This sections provides additional information about the inverse tangent function. For further details, see the supplementary paper [9] cited in Section 2.10 “References” on page 126.

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

Mathematical definition:

$$\operatorname{atan2}(Y,\ X) \supseteq \bigcup_{\substack{x \in X \\ y \in Y}} \{\theta \mid h\sin\theta = y,\ h\cos\theta = x, h = (x^2 + y^2)^{1/2}\}$$

Class: Elemental function.

Special values: TABLE 2-14 and CODE EXAMPLE 2-54 display the `ATAN2` indeterminate forms.

TABLE 2-14 `ATAN2` Indeterminate Forms

y_0	x_0	cset(sinθ, {y_0, x_0})	cset(cosθ, {y_0, x_0})	cset(θ, {y_0, x_0})
0	0	[-1, 1]	[-1, 1]	$[-\pi,\ \pi]$
+∞	+∞	[0, 1]	[0, 1]	$[0,\ \frac{\pi}{2}]$
+∞	-∞	[0, 1]	[-1, 0]	$[\frac{\pi}{2},\ \pi]$
-∞	-∞	[-1, 0]	[-1, 0]	$[-\pi,\ \frac{-\pi}{2}]$
-∞	+∞	[-1, 0]	[0, 1]	$[\frac{-\pi}{2},\ 0]$

CODE EXAMPLE 2-54 `ATAN2` Indeterminate Forms

```
math% cat ce2-54.f95
   INTERVAL :: X, Y
   INTEGER  :: IOS = 0
   PRINT *, "Press Control/D to terminate!"
   WRITE(*, 1, ADVANCE='NO')
   READ(*, *, IOSTAT=IOS) Y, X
   DO WHILE (ios >= 0)
      PRINT *, "For X =", X, "For Y =", Y
      PRINT *, 'ATAN2(Y,X)= ', ATAN2(Y,X)
      WRITE(*, 1, ADVANCE='NO')
      READ(*, *, IOSTAT=IOS) Y, X
   END DO
1  FORMAT("Y, X = ?")
   END
```

CODE EXAMPLE 2-54 ATAN2 Indeterminate Forms *(Continued)*

```
math% f95 -xia ce2-54.f95
math% a.out
 Press Control/D to terminate!
Y, X = ? [0] [0]
For X = [0.0E+0,0.0E+0] For Y = [0.0E+0,0.0E+0]
ATAN2(Y,X)=  [-3.1415926535897936,3.1415926535897936]
 Y, X = ? inf inf
For X = [1.7976931348623157E+308,Inf] For Y =
[1.7976931348623157E+308,Inf]
 ATAN2(Y,X)=  [0.0E+0,1.5707963267948968]
 Y, X = ?inf -inf
For X = [-Inf,-1.7976931348623157E+308] For Y =
 [1.7976931348623157E+308,Inf]
 ATAN2(Y,X)=  [1.5707963267948965,3.1415926535897936]
 Y, X = ?-inf +inf
For X = [1.7976931348623157E+308,Inf] For Y =
[-Inf,-1.7976931348623157E+308]
 ATAN2(Y,X)=  [-1.5707963267948968,0.0E+0]
 Y, X = ?-inf -inf
For X = [-Inf,-1.7976931348623157E+308] For Y =
[-Inf,-1.7976931348623157E+308]
 ATAN2(Y,X)=  [-3.1415926535897936,-1.5707963267948965]
 Y, X = ? <Control-D>
```

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 an interval of minimum width that contains the exact mathematical interval in the description.

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

In the case where $\overline{x} < 0$ and $0 \in Y$, to get a sharp interval enclosure (denoted by Θ), the following convention uniquely defines the set of all possible returned interval angles:

$$-\pi < m(\Theta) \le \pi$$

This convention, together with

$$0 \le w(\Theta) \le 2\pi$$

results in a unique definition of the interval angles Θ that ATAN2(Y, X) must include.

TABLE 2-15 contains the tests and arguments of the `REAL ATAN2` function that are used to compute the endpoints of Θ in the algorithm that satisfies the constraints required to produce sharp interval angles. The first two columns define the distinguishing cases. The third column contains the range of possible values of the midpoint, *m*(Θ), of the interval Θ. The last two columns show how the endpoints of Θ are computed using the `REAL ATAN2` intrinsic function. Directed rounding must be used to guarantee containment.

TABLE 2-15 Tests and Arguments of the `REAL ATAN2` Function

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

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

Description: Range of maximum.

The containment set for max(X_1, ..., X_n) is:

$$\{z \mid z = \max(x_1, \ldots, x_n), x_i \in X_i\} = [\sup(\text{hull}(\underline{x}_1, \ldots, \underline{x}_n)), \sup(\text{hull}(\overline{x}_1, \ldots, \overline{x}_n))] .$$

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 have the same type and `KIND` type parameter.

Result characteristics: The result is of type `INTERVAL`. The kind type parameter is that of the arguments.

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

Description: Range of minimum.

The containment set for min(X_1, ..., X_n) is:

$$\{z \mid z = \min(x_1, \ldots, x_n), x_i \in X_i\} = [\inf(\text{hull}(\underline{x}_1, \ldots, \underline{x}_n)), \inf(\text{hull}(\overline{x}_1, \ldots, \overline{x}_n))] .$$

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 have the same type and `KIND` type parameter.

Result characteristics: The result is of type `INTERVAL`. The kind type parameter is that of the arguments.

2.9.4.4 Intrinsic Functions

Tables TABLE 2-17 through TABLE 2-21 list the properties of intrinsic functions that accept interval arguments. TABLE 2-16 lists the tabulated properties of intrinsic `INTERVAL` functions in these tables.

TABLE 2-16 Tabulated Properties of Each Intrinsic `INTERVAL` Function

Tabulated Property	Description
Intrinsic Function	what the function does
Definition	mathematical definition
No. of Args.	number of arguments the function accepts
Generic Name	the function's generic name
Type-Specific Names	the function's specific names
Argument Type	data type associated with each specific name
Function Type	data type returned for specific argument data type

KTPV 4, 8 and 16 versions of intrinsic `INTERVAL` functions are defined. The corresponding specific intrinsic names begin with `VS`, `VD` or `VQ`, from `interVal` **S**ingle, **D**ouble and **Q**uad.

For each specific `REAL` intrinsic function, a corresponding intrinsic `INTERVAL` function exists with a `VS`, `VD,` or `VQ` prefix, such as `VSSIN()` and `VDSIN()`.

Because indeterminate forms are possible, special values of the X**Y and ATAN2 function are contained in Section 2.5 "Power Operators `X**N` and `X**Y`" on page 60 and Section 2.9.4.1 "Inverse Tangent Function `ATAN2(Y,X)`" on page 117, respectively. The remaining intrinsic functions do not require this treatment.

TABLE 2-17 Intrinsic `INTERVAL` Arithmetic Functions

Intrinsic Function	Point Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Absolute value	|*a*|	1	`ABS`	`VDABS`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSABS`	`INTERVAL(4)`	`INTERVAL(4)`
				`VQABS`	`INTERVAL(16)`	`INTERVAL(16)`
Truncation *See Note 1*	int(*a*)	1	`AINT`	`VDINT`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSINT`	`INTERVAL(4)`	`INTERVAL(4)`
				`VQINT`	`INTERVAL(16)`	`INTERVAL(16)`
Nearest integer	int(*a* + .5) if $a \geq 0$ int(*a* - .5) if $a < 0$	1	`ANINT`	`VDNINT`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSNINT`	`INTERVAL(4)`	`INTERVAL(4)`
				`VQNINT`	`INTERVAL(16)`	`INTERVAL(16)`
Remainder	*a*-*b*(int(*a*/*b*))	2	`MOD`	`VDMOD`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSMOD`	`INTERVAL(4)`	`INTERVAL(4)`
Transfer of sign *See Note 2*	|*a*| sgn(*b*)	2	`SIGN`	`VDSIGN`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSSIGN`	`INTERVAL(4)`	`INTERVAL(4)`
Choose largest value *See Note 3*	max(*a*,*b*,...)	≥2	`MAX`	`MAX`	`INTERVAL`	`INTERVAL`
Choose smallest value *See Note 3*	min(*a*,*b*,...)	≥2	`MIN`	`MIN`	`INTERVAL`	`INTERVAL`

(1) int(*a*) = floor(*a*) if a > 0 and ceiling(*a*) if a < 0

(2) The signum function sgn(*a*) = -1 if *a* < 0, +1 if *a* > 0 and 0 if *a* = 0

(3) The `MIN` and `MAX` intrinsic functions ignore empty interval arguments unless all arguments are empty, in which case, the empty interval is returned.

TABLE 2-18 Intrinsic `INTERVAL` Type Conversion Functions

Conversion To	No. of Args.	Generic Name	Argument Type	Function Type
INTERVAL	1, 2, or 3	`INTERVAL`	`INTERVAL`	`INTERVAL`
			`INTERVAL(4)`	`INTERVAL`
			`INTERVAL(8)`	`INTERVAL`
			`INTEGER`	`INTERVAL`
			`REAL`	`INTERVAL`
			`REAL(8)`	`INTERVAL`
			`REAL(16)`	`INTERVAL`
INTERVAL(4)	1 or 2	`SINTERVAL`	`INTERVAL`	`INTERVAL(4)`
			`INTERVAL(4)`	`INTERVAL(4)`
			`INTERVAL(8)`	`INTERVAL(4)`
			`INTEGER`	`INTERVAL(4)`
			`REAL`	`INTERVAL(4)`
			`REAL(8)`	`INTERVAL(4)`
			`REAL(16)`	`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)`
			`REAL(8)`	`INTERVAL(8)`
			`REAL(16)`	`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)`
			`REAL(8)`	`INTERVAL(16)`

TABLE 2-19 Intrinsic `INTERVAL` Trigonometric Functions

Intrinsic Function	Point Definition	No. of Args.	Generic Name	Specific Names	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)`
Arctangent	arctan(*a/b*)	2	`ATAN2`	`VDATAN2`	`INTERVAL(8)`	`INTERVAL(8)`
See Note 1				`VSATAN2`	`INTERVAL(4)`	`INTERVAL(4)`
Hyperbolic	sinh(*a*)	1	`SINH`	`VDSINH`	`INTERVAL(8)`	`INTERVAL(8)`
Sine				`VSSINH`	`INTERVAL(4)`	`INTERVAL(4)`
Hyperbolic	cosh(*a*)	1	`COSH`	`VDCOSH`	`INTERVAL(8)`	`INTERVAL(8)`
Cosine				`VSCOSH`	`INTERVAL(4)`	`INTERVAL(4)`
Hyperbolic	tanh(*a*)	1	`TANH`	`VDTANH`	`INTERVAL(8)`	`INTERVAL(8)`
Tangent				`VSTANH`	`INTERVAL(4)`	`INTERVAL(4)`

(1) arctan(a/b) = θ, given $a = h \sin\theta$, $b = h \cos\theta$, and $h^2 = a^2 + b^2$.

TABLE 2-20 Other Intrinsic `INTERVAL` Mathematical Functions

Intrinsic Function	Point Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
Square Root *See Note 1*	$\exp\{\ln(a)/2\}$	1	`SQRT`	`VDSQRT`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSSQRT`	`INTERVAL(4)`	`INTERVAL(4)`
Exponential	$\exp(a)$	1	`EXP`	`VDEXP`	`INTERVAL`	`INTERVAL(8)`
				`VSEXP`	`INTERVAL(4)`	`INTERVAL(4)`
Natural logarithm	$\ln(a)$	1	`LOG`	`VDLOG`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSLOG`	`INTERVAL(4)`	`INTERVAL(4)`
Common logarithm	$\log(a)$	1	`LOG10`	`VDLOG10`	`INTERVAL(8)`	`INTERVAL(8)`
				`VSLOG10`	`INTERVAL(4)`	`INTERVAL(4)`

(1) sqrt(a) is multi-valued. A proper interval enclosure must contain both the positive and negative square roots. Defining the `SQRT` intrinsic to be

$$\exp\left\{\frac{\ln a}{2}\right\}$$

eliminates this difficulty.

TABLE 2-21 Intrinsic `INTERVAL`-Specific Functions

Intrinsic Function	Definition	No. of Args.	Generic Name	Specific Names	Argument Type	Function Type
INF	inf([a, b]) = a	1	`INF`	`VDINF`	`INTERVAL(8)`	`REAL(8)`
				`VSINF`	`INTERVAL(4)`	`REAL(4)`
				`VQINF`	`INTERVAL(16)`	`REAL(16)`
SUP	sup([a, b]) = b	1	`SUP`	`VDSUP`	`INTERVAL(8)`	`REAL(8)`
				`VSSUP`	`INTERVAL(4)`	`REAL(4)`
				`VQSUP`	`INTERVAL(16)`	`REAL(16)`
Width	w([a, b]) = b - a	1	`WID`	`VDWID`	`INTERVAL(8)`	`REAL(8)`
				`VSWID`	`INTERVAL(4)`	`REAL(4)`
				`VQWID`	`INTERVAL(16)`	`REAL(16)`
Midpoint	mid([a, b]) = (a + b)/2	1	`MID`	`VDMID`	`INTERVAL(8)`	`REAL(8)`
				`VSMID`	`INTERVAL(4)`	`REAL(4)`
				`VQMID`	`INTERVAL(16)`	`REAL(16)`
Magnitude *See Note 1*	max($\|a\|$) $\in A$	1	`MAG`	`VDMAG`	`INTERVAL(8)`	`REAL(8)`
				`VSMAG`	`INTERVAL(4)`	`REAL(4)`
				`VQMAG`	`INTERVAL(16)`	`REAL(16)`
Mignitude *See Note 2*	min($\|a\|$) $\in A$	1	`MIG`	`VDMIG`	`INTERVAL(8)`	`REAL(8)`
				`VSMIG`	`INTERVAL(4)`	`REAL(4)`
				`VQMIG`	`INTERVAL(16)`	`REAL(16)`
Test for empty interval	*true* if A is empty	1	`ISEMPTY`	`VDISEMPTY`	`INTERVAL(8)`	`LOGICAL`
				`VSISEMPTY`	`INTERVAL(4)`	`LOGICAL`
				`VQISEMPTY`	`INTERVAL(16)`	`LOGICAL`
Floor	floor(A)	1	`FLOOR`		`INTERVAL(8)`	`INTEGER`
					`INTERVAL(4)`	`INTEGER`
					`INTERVAL(16)`	`INTEGER`
Ceiling	ceiling(A)	1	`CEILING`		`INTERVAL(8)`	`INTEGER`
					`INTERVAL(4)`	`INTEGER`
					`INTERVAL(16)`	`INTEGER`
Precision	precision(A)	1	`PRECISION`		`INTERVAL(8)`	`INTEGER`
					`INTERVAL(4)`	`INTEGER`
					`INTERVAL(16)`	`INTEGER`
Range	range(A)	1	`RANGE`		`INTERVAL(8)`	`INTEGER`
					`INTERVAL(4)`	`INTEGER`
					`INTERVAL(16)`	`INTEGER`
Number of digits *See Note 3*	Maximum number of digits using `Y` edit descriptor	1	`NDIGITS`		`INTERVAL`	`INTEGER`
					`INTERVAL(4)`	`INTEGER`
					`INTERVAL(16)`	`INTEGER`

(1) mag([a, b]) = max($|a|$,$|b|$)

(2) mig([a, b]) = min($|a|$,$|b|$), if $a > 0$ or $b < 0$, otherwise 0

(3) Special cases: `NDIGITS([-inf, +inf])` = `NDIGITS([EMPTY])` = 0

2.10 References

The following technical reports are available online. See the Interval Arithmetic README for the location of these files.

1. G.W. Walster, E.R. Hansen, and J.D. Pryce, “Extended Real Intervals and the Topological Closure of Extended Real Relations,” Technical Report, Sun Microsystems. February 2000.
2. G. William Walster, “Empty Intervals,” Technical Report, Sun Microsystems. April 1998.
3. G. William Walster, “Closed Interval Systems,” Technical Report, Sun Microsystems. August 1999.
4. G. William Walster, “Literal Interval Constants,” Technical Report, Sun Microsystems. August 1999.
5. G. William Walster, “Widest-Need Interval Expression Evaluation,” Technical Report, Sun Microsystems. August 1999.
6. G. William Walster, “Compiler Support of Interval Arithmetic With Inline Code Generation and Nonstop Exception Handling,” Technical Report, Sun Microsystems. February 2000.
7. G. William Walster, “Finding Roots on the Edge of a Function’s Domain,” Technical Report, Sun Microsystems. February 2000.
8. G. William Walster, “Implementing the ‘Simple’ Closed Interval System,” Technical Report, Sun Microsystems. February 2000.
9. G. William Walster, “Interval Angles and the Fortran ATAN2 Intrinsic Function,” Technical Report, Sun Microsystems. February 2000.
10. G. William Walster, “The ‘Simple’ Closed Interval System,” Technical Report, Sun Microsystems. February 2000.
11. G. William Walster, Margaret S. Bierman, “Interval Arithmetic in Forte Developer Fortran,” Technical Report, Sun Microsystems. March 2000.

Glossary

affirmative relation	An order relation other than certainly, possibly, or set not equal. *Affirmative relations* affirm something, such as $a < b$.
affirmative relational operators	An *affirmative relational operator* is an element of the set: $\{<, \leq, =, \geq, >\}$.
anti-affirmative relation	An *anti-affirmative relation* is a statement about what cannot be true. The order relation $\neq$ is the only anti-affirmative relation in Fortran.
anti-affirmative relational operator	The Fortran `.NE.` and `/=` operators implement the anti-affirmative relation. The certainly, possible, and set versions for interval operands are denoted `.CNE.`, `.PNE.`, and `.SNE.`, respectively.
assignment statement	An *assignment statement* is a Fortran statement having the form: `V =` *`expression.`* The left-hand side of the assignment statement is the variable, array element, or array, `V`.
certainly true relational operator	See ***relational operators: certainly true***.
closed interval	A *closed interval* includes its endpoints. A closed interval is a *closed set*. The interval $[2, 3] = \{z \mid 2 \leq z \leq 3\}$ is closed, because its endpoints are included. The interval $(2, 3) = \{z \mid 2 < z < 3\}$ is open, because its endpoints are not included. Interval arithmetic, as implemented in `f95`, only deals with closed intervals.
closed mathematical system	In a *closed mathematical system*, there can be no undefined operator-operand combinations. All defined operations on elements of a closed system must produce elements of the system. The real number system is not closed, because division by zero is undefined in this system.

closed set A *closed set* contains all limit or accumulation points in the set. That is, given the set, S, and sequences, $\{s_j\} \in S$, the closure of S is $\bar{S} = \{\lim_{j \to \infty} s_j \mid s_j \in S\}$, where $\lim_{j \to \infty}$ denotes an accumulation or limit point of the sequence $\{s_j\}$.

The set of real numbers is the open set $\{z \mid -\infty < z < +\infty\}$, because it does not include $-\infty$ and $+\infty$. The set of extended real numbers, $\Re^*$, is closed.

closure-composition equality Given the expressions f, g, and h, with

$$f(\{x_0\}) = g(\{(y, x_0) \mid y = h(\{x_0\})\}),$$

the closure-composition equality states that

$$\bar{f}(\{x_0\}) = \bar{g}(\{(y, x_0) \mid y = \bar{h}(\{x_0\})\}).$$

The closure of $\bar{f}$ at the point x_0 is equal to the composition of its component's closures, $\bar{g}$ and $\bar{h}$.

closure of expression The closure of the expression f *of* n-variables, evaluated over the set $\boldsymbol{X}_0$ is denoted $\bar{f}(\boldsymbol{X}_0)$, and for $x_0 \in \bar{D}_f$ is defined:

$$\bar{f}(\boldsymbol{X}_0) = \left\{ z \;\middle|\; \begin{array}{l} z = \lim_{j \to \infty} y_j \\ y_j \in f(\{\boldsymbol{x}_j\}) \\ \boldsymbol{x}_j \in D_f \\ \lim_{j \to \infty} \boldsymbol{x}_j \in \boldsymbol{X}_0 \end{array} \right\}$$

For $\boldsymbol{X}_0 \notin \bar{D}_f$, $\bar{f}(\boldsymbol{X}_0) = \varnothing$.

The accumulation points, $\lim_{j \to \infty} x_j$, of all sequences, $\{\boldsymbol{x}_j\}$, are elements of the set, $\boldsymbol{X}_0$. The closure of f is the set of all possible accumulation points of f given the conditions on the right-hand side of the above defining expression are satisfied.

$\bar{f}(\boldsymbol{X}_0)$ is defined for all $\boldsymbol{X}_0 \in (\Re^*)^n$.

connected set The *connected set* of numbers between and including two values, $a \le b$, contains all the values between and including a and b.

composite expression Forming a new expression, f, (the *composite expression*) from the given expressions, g and h by the rule $f(\{\underline{x}\}) = g(h(\{\underline{x}\}))$ for all singleton sets, $\{\underline{x}\} = \{x_1\} \otimes \ldots \otimes \{x_n\}$ in the domain of h for which h is in the domain of g. Singleton set arguments connote the fact that expressions can be either functions or relations.

constant expression A *constant expression* in Fortran contains no variables or arrays. It can contain constants and operands. The expression `[2, 3] + [4, 5]` is a constant expression. If `X` is a variable, the expression `X + [2, 3]` is not a constant expression. If `Y` is a named constant, `Y + [2, 3]` is a constant expression.

containment constraint The *containment constraint* on the interval evaluation, $f([x])$, of the expression, f, at the degenerate interval, $[x]$, is:

$f([x]) \supseteq \text{cset}(f,\{x\})$,

where $\text{cset}(f,\{x\})$ denotes the containment set of all possible values that $f([x])$ must contain. Because the containment set, $\text{cset}(x \div y, \{(1, 0)\}) = \{-\infty, +\infty\}$, $[1] / [0] = \text{hull}(\{-\infty, +\infty\}) = [-\infty, +\infty]$. See also ***containment set***.

containment failure A *containment failure* is a failure to satisfy the containment constraint. For example, a containment failure results if [1]/[0] is defined to be [*empty*]. This can be seen by considering the interval expression

$$\frac{X}{X+Y} = \frac{1}{1+\frac{Y}{X}}$$

for X=[0] and Y, given $0 \notin Y$. The containment set of the first expression is [0]. However, if [1]/[0] is defined to be [*empty*], the second expression is also [*empty*]. This is a containment failure.

containment set The *containment set*, $\text{cset}(h, \{x\})$, of the expression h is the smallest set that does not violate the containment constraint when h is used as a component of any composition, $f(\{x\}) = g(h(\{x\}), \{x\})$.

For $h(x, y) = x \div y$,

$\text{cset}(h, \{(+\infty, +\infty)\}) = [0, +\infty]$.

See also ***cset(expression, set)***.

containment set closure identity Given any expression $f(\{\boldsymbol{x}\}) = f(\{x_1\} \otimes \ldots \otimes \{x_n\})$ of n-variables and the point, $\boldsymbol{x}_0$, then $\text{cset}(f, \{\boldsymbol{x}_0\}) = \bar{f}(\{\boldsymbol{x}_0\})$, the closure of f at the point, $\boldsymbol{x}_0$.

containment set equivalent Two expressions are *containment-set equivalent* if their containment sets are everywhere identical.

containment set evaluation theorem Let $\overline{\text{eval}}(f, \{\boldsymbol{x}\})$ denote the code-list evaluation of the expression f, using individual composition closures to compute the value of every sub-expression, whether it is the value of an operation, function, or relation. Given the expression, $f(\{\boldsymbol{x}\}) = f(\{x_1\} \otimes \ldots \otimes \{x_n\})$, whether f is a function or a relation, then for all $\boldsymbol{x}_0 \in (\Re^*)^n$, $\text{cset}(f, \{\boldsymbol{x}_0\}) \subseteq \overline{\text{eval}}(f, \{\boldsymbol{x}_0\})$.

context-dependent `INTERVAL` constant The internal approximation of an `INTERVAL` constant under widest-need expression processing is *context dependent*, because it is a sharp interval with KTPV that equals KTPV$_{\text{max}}$. Any approximation for the interval constant [*a*, *b*] can be used, provided,

$[a, b] \supseteq \text{ev}([a, b])$,

where ev([*a*, *b*]) denotes the external value of the interval constant, [a, *b*]. Choosing any internal approximation is permitted, provided containment is not violated. For example, the internal approximations, `[0.1_4]`, `[0.1_8]`, and `[0.1_16]`, all have external value, ev(`0.1`) = 1/10, and therefore do not violate the containment constraint. Under widest-need expression processing the internal approximation is used that has the same KTPV as KTPV$_{\text{max}}$.

cset(*expression*, *set*) The notation, cset(*expression*, *set*), is used to symbolically represent the containment set of an expression evaluated over a set of arguments. For example, for the expression, $f(x, y) = xy$, the containment constraint that the interval expression $[0] \times [+\infty]$ must satisfy is

$[0] \times [+\infty] \supseteq \text{cset}(x \times y, \{(0,+\infty)\}) = [-\infty, +\infty]$.

degenerate interval A *degenerate interval* is a zero-width interval. A degenerate interval is a singleton set, the only element of which is a point. In most cases, a degenerate interval can be thought of as a point. For example, the interval [2, 2] is degenerate, and the interval [2, 3] is not.

directed rounding *Directed rounding* is rounding in a particular direction. In the context of interval arithmetic, rounding up is towards +∞, and rounding down is towards -∞. The direction of rounding is symbolized by the arrows, $\downarrow$ and $\uparrow$. Therefore, with 5-digit arithmetic, $\uparrow 2.00001 = 2.0001$. Directed rounding is used to implement interval arithmetic on computers so that the containment constraint is never violated.

disjoint interval Two *disjoint intervals* have no elements in common. The intervals [2, 3] and [4, 5] are disjoint. The intersection of two disjoint intervals is the empty interval.

empty interval The *empty interval,* [*empty*], is the interval with no members. The empty interval naturally occurs as the intersection of two disjoint intervals. For example, $[2, 3] \cap [4,5] =$ [*empty*].

empty set The *empty set*, ∅, is the set with no members. The empty set naturally occurs as the intersection of two disjoint sets. For example, $\{2, 3\} \cap \{4, 5\} = \emptyset$.

ev(*`literal_constant`*) The notation ev(*`literal_constant`*) is used to denote the external value defined by a literal constant character string. For example, ev(`0.1`) = 1/10, in spite of the fact that an internal approximation of 0.1 must be used, because the constant 0.1 is not machine representable.

exception In the IEEE 754 floating-point standard, an *exception* occurs when an attempt is made to perform an undefined operation, such as division by zero.

exchangeable expression	Two expressions are exchangeable if they are containment-set equivalent (their containment sets are everywhere identical).
expression context	In widest-need expression processing, the two attributes that define *expression context* are the expression's type and the maximum KTPV (KTPV_{max}).
expression processing: strict	See **strict expression processing**.
expression processing: widest-need	See ***widest-need expression processing***.
extended interval	The term *extended interval* refers to intervals whose endpoints can be extended real numbers, including $-\infty$ and $+\infty$. For completeness, the empty interval is also included in the set of extended real intervals.
external representation	The *external representation* of a Fortran data item is the character string used to define it during input data conversion, or the character string used to display it after output data conversion.
external value	The *external value* of a Fortran literal constant is the mathematical value defined by the literal constant's character string. The external value of a literal constant is not necessarily the same as the constant's internal approximation, which, in the Fortran standard, is the only defined value of a literal constant. See ***ev(literal_constant)***.
hull	See ***interval hull***.
infimum (plural, infima)	The *infimum* of a set of numbers is the set's greatest lower bound. This is either the smallest number in the set or the largest number that is less than all the numbers in the set. The infimum, $\inf([a, b])$, of the interval constant $[a, b]$ is a.
interval algorithm	An *interval algorithm* is a sequence of operations used to compute an interval result.
internal approximation	In Fortran, the *internal approximation* of a literal constant is a machine representable value. There is no internal approximation accuracy requirement in the Fortran standard.
interval arithmetic	*Interval arithmetic* is the system of arithmetic used to compute with intervals.
interval box	An interval box is a parallelepiped with sides parallel to the n-dimensional Cartesian coordinate axes. An interval box is conveniently represented using an n-dimensional interval vector, $\boldsymbol{X} = (X_1, \ldots, X_n)^{\mathrm{T}}$.
`INTERVAL` constant	An *`INTERVAL` constant* is the closed corrected set: $[a, b] = \{z \mid a \le z \le b\}$ defined by the pair of numbers, $a \le b$.

`INTERVAL` constant's external value	An *`INTERVAL` constant's external value* is the mathematical value defined by the interval constant's character string. See also ***external value***.
`INTERVAL` constant's internal approximation	In `f95`, an *`INTERVAL` constant's internal approximation* is the sharp internal approximation of the constant's external value. Therefore, it is the narrowest possible machine representable interval that contains the constant's external value.
interval hull	The *interval hull* operator, $\underline{\cup}$, on a pair of intervals $X = [\bar{x}, \underline{x}]$ and $Y = [\bar{y}, \underline{y}]$, is the smallest interval that contains both X and Y (also represented as $[\inf (X \cup Y),\ \sup (X \cup Y)]$). For example, $[2, 3] \ \underline{\cup}\ [5, 6] = [2, 6]$.
`INTERVAL`-specific function	In `f95`, an *`INTERVAL`-specific function* is an interval function that is not an interval version of a standard Fortran function. For example, `WID`, `MID`, `INF`, and `SUP`, are `INTERVAL`-specific functions.
interval width	Interval width, $w([a, b]) = b - a$.
intrinsic `INTERVAL` data type	In Fortran, there are four intrinsic numeric data types: `INTEGER`, `REAL`, `DOUBLE PRECISION REAL`, and `COMPLEX`. With the command line option `-xia` or `-xinterval`, `f95` recognizes `INTERVAL` as an intrinsic data type.
intrinsic `INTERVAL`-specific function	In `f95`, there are a variety of *intrinsic `INTERVAL`-specific functions*, including: `WID`, `HULL`, `MID`, `INF`, and `SUP`.
kind type parameter value (KTPV)	In Fortran, each intrinsic data type is parameterized using a *kind type parameter value (KTPV),* which selects the kind (precision) of the data type. In `f95`, there are three `INTERVAL` KTPVs: 4, 8, and 16. The default interval KTPV is 8.
KTPV (kind type parameter value)	See ***kind type parameter value (KTPV)***.
KTPV_{max}	In widest-need expression processing of interval expressions, all intervals are converted to the maximum value of the KTPV of any data item in the expression. This maximum value is given the name KTPV_{max}.
left endpoint	The *left endpoint* of an interval is the same as its infimum or lower bound.
literal constant	In `f95`, an *interval literal constant* is the character string used to define the constant's external value.

literal constant's external value In `f95`, an *interval literal constant's external value* is the mathematical value defined by the constant's character string. See also ***external value***.

literal constant's internal approximation In `f95`, an *interval literal constant's internal approximation* is the sharp machine representable interval that contains the constant's external value.

lower bound See ***infimum (plural, infima)***.

mantissa When written in scientific notation, a number consists of a *mantissa* or significand and an exponent power of 10. The `E` edit descriptor in Fortran displays numbers in terms of a mantissa or significand and an exponent, or power of 10.

mixed-KTPV `INTERVAL` expression A *mixed-KTPV `INTERVAL` expression* contains constants and/or variables with different KTPVs. For example, `[1_4] + [0.2_8]` is a mixed-KTPV `INTERVAL` expression. Mixed-KTPV interval expressions are permitted under widest-need expression processing, but are not permitted under strict expression processing.

mixed-mode (type and KTPV) `INTERVAL` expression A *mixed-mode `INTERVAL` expression* contains data items of different types and KTPV. For example, the expression `[0.1] + 0.2` is a mixed-mode expression. `[0.1]` is an `INTERVAL` constant with KTPV = 8, while `0.2` is a `REAL` constant with KTPV = 4.

mixed-type `INTERVAL` expression A *mixed-type `INTERVAL` expression* contains data items of different types. For example, the expression `[0.1] + 0.2D0` is a mixed-type `INTERVAL` expression, because `[0.1]` is an `INTERVAL`, and `0.2D0` is a `DOUBLE PRECISION` constant. They both have the same KTPV = 8.

multiple-use expression (MUE) A *multiple-use expression (MUE)* is an expression in which at least one independent variable appears more than once.

named constant A *named constant* is declared and initialized in a `PARAMETER` statement. Because the value of a named constant is not context dependent, a more appropriate name for a data item in a `PARAMETER` declaration is "read-only variable."

narrow-width interval Let the interval $[a, b]$ be an approximation of the value $\nu \in [a, b]$. If $w[a, b] = b - a$, is small, $[a, b]$ is a *narrow-width interval*. The narrower the width of the interval $[a, b]$, the more accurately $[a, b]$ approximates ν. See also ***sharp interval result***.

opaque data type An *opaque data type* leaves the structure of internal approximations unspecified. `INTERVAL` data items are opaque. Therefore, programmers cannot count on `INTERVAL` data items being internally represented in any particular way. The intrinsic functions `INF` and `SUP` provide access to the components of an interval. The `INTERVAL` constructor can be used to manually construct any valid interval.

point A *point* (as opposed to an interval), is a number. A point in n-dimensional space, is represented using an n-dimensional vector, $\boldsymbol{x} = (x_1, \ldots, x_n)^T$. A point and a degenerate interval, or interval vector, can be thought of as the same. Strictly, any interval is a set, the elements of which are points.

possibly true relational operators See ***relational operators: possibly true***.

quality of implementation *Quality of implementation*, is a phrase used to characterize properties of compiler support for intervals. Narrow width is a new quality of implementation opportunity provided by intrinsic compiler support for `INTERVAL` data types.

radix conversion *Radix conversion* is the process of converting back and forth between external decimal numbers and internal binary numbers. Radix conversion takes place in formatted and list-directed input/output. Because the same numbers are not always representable in the binary and decimal number systems, guaranteeing containment requires directed rounding during radix conversion.

read-only variable A *read-only variable* is not a defined construct in standard Fortran. Nevertheless, a read-only variable is a variable, the value of which cannot be changed once it is initialized. In standard Fortran, without interval support, there is no need to distinguish between a named constant and a read-only variable. Because widest-need expression processing uses the external value of constants, the distinction between a read-only variable and a named constant must be made. As implemented in `f95`, the symbolic name that is initialized in a `PARAMETER` declaration is a read-only variable.

relational operators: certainly true The *certainly true relational operators* are {`.CLT.`, `.CLE.`, `.CEQ.`, `.CNE.`, `.CGE.`, `.CGT.`}. Certainly true relational operators are true if the relation in question is true for all elements in the operand intervals. That is $[a, b]$ .Cop. $[c, d]$ = *true* if x .op. y = *true* for all $x \in [a, b]$ and $y \in [c, d]$.

For example, $[a, b]$ `.CLT.` $[c, d]$ if $b < c$.

relational operators: possibly true The *possibly true relational operators* are {`.PLT.`, `.PLE.`, `.PEQ.`, `.PNE.`, `.PGE.`, `.PGT.`}. Possibly true relational operators are true if the relation in question is true for any elements in operand intervals. For example, $[a, b]$ `.PLT.` $[c, d]$ if $a < d$.

relational operators: set

The *set relational operators* are {.SLT., .SLE., .SEQ., .SNE., .SGE., .SGT.}. Set relational operators are true if the relation in question is true for the endpoints of the intervals. For example, [*a*, *b*] .SEQ. [*c*, *d*] if ($a = c$) and ($b = d$).

right endpoint

See ***supremum (plural, suprema)***.

scope of widest-need expression processing

See ***widest-need expression processing: scope***.

set theoretic

Set theoretic is the means of or pertaining to the algebra of sets.

sharp interval result

A *sharp interval result* has a width that is as narrow as possible. A sharp interval result is equal to the hull of an expression's containment. Given the limitations imposed by a particular finite precision arithmetic, a sharp interval result is the narrowest possible finite precision interval that contains the expression's containment set.

single-number input/output

Single-number input/output, uses the single-number external representation for an interval, in which the interval $[-1, +1]_{\text{uld}}$ is implicitly added to the last displayed digit. The subscript *uld* is an acronym for unit in the last digit. For example 0.12300 represents the interval $0.12300 + [-1, +1]_{\text{uld}} = [0.12299, 0.12301]$.

single-number `INTERVAL` data conversion

Single-number `INTERVAL` data conversion is used by the `Y` edit descriptor to read and display external intervals using the single-number representation. See ***single-number input/output***.

single-use expression (SUE)

A *single-use expression (SUE)* is an expression in which each variable only occurs once. For example

$$\frac{1}{1 + \frac{Y}{X}}$$

is a single use expression, whereas

$$\frac{X}{X + Y}$$

is not.

strict expression processing

Under *strict expression processing*, no automatic type or KTPV changes are made by the compiler. Mixed type and mixed KTPV `INTERVAL` expressions are not allowed. Any type and/or KTPV changes must be explicitly programmed.

supremum (plural, suprema)	The *supremum* of a set of numbers is the set's least upper bound. This is either the largest number in the set or the smallest number that is greater than all the numbers in the set. The supremum, sup([a, b]), of the interval constant [a, b] is b.
unit in the last digit (uld)	In single number input/output, one *unit in the last digit (uld)* is added to and subtracted from the last displayed digit to implicitly construct an interval.
unit in the last place (ulp)	One *unit in the last place (ulp)* of an internal machine number is the smallest possible increment or decrement that can be made using the machine's arithmetic. Therefore, if the width of a computed interval is 1-ulp, this is the narrowest possible non-degenerate interval with a given KTPV.
upper bound	See ***supremum (plural, suprema)***.
valid interval result	A *valid interval result*, [a, b] must satisfy two requirements: ■ $a \le b$ ■ [a, b] must not violate the containment constraint
value assignment	In Fortran, an assignment statement computes the value of the expression to the right of the assignment of value operator, =, and stores the value in the variable, array element, or array to the left of the assignment of value operator.
widest-need expression processing	Under *widest-need expression processing*, automatic type and KTPV changes are made by the compiler. Any non-interval subexpressions are promoted to intervals and KTPVs are set to KTPV_{max}.
widest-need expression processing: scope	In Fortran, scope refers to that part of an executable program where data and/or operations are defined and unambiguous. The scope of widest-need expression processing is limited by calls to functions and subroutines.

Index