Expression Reference Mar. 17, 2025

Expressions have a structural representation, which can be evaluated or manipulated symbolically, and a rendered representation, which is subject to style choices. The rendered representation supports cursor movement, selection ranges, and editing. An expression can be constructed in code then rendered or constructed interactively through a visual editor and then evaluated. For example, a student could enter a function visually which could then be graphed or tested against a “correct” function by evaluating at several test points.

**Expression Trees**

An expression is a **tree** (directed acyclic graph) of **nodes** in which every node can be evaluated to a well-defined data type. The root node’s value is the value of the expression. Evaluating a tree node involves first evaluating all child nodes then using their values to compute the node’s value.

Evaluation may generate errors (for example, an attempt to divide by zero or evaluate an even root of a negative number when complex values are not allowed). After evaluation, every node has an “error” flag that will be set if the node generated an error, allowing those nodes to be highlighted in visual representations.

**Evaluation Context**

Evaluation depends on an **evaluation context** which contains values of variables referenced in the expression and evaluating settings (such as whether complex values are allowed).

The evaluation context can be set to allow complex results or to force real results. If complex results are disallowed, any result that has a nonzero imaginary part will be replaced with **Error**.

**Variable Names and Representations**

Every variable has a unique name that may include superscripts or subscripts and a display style (the combination of all these forms the unique variable identifier). These are authored as simple strings but can be rendered with proper mathematical notation and style.

The syntax for variables is

%presuperscript\_presubscript%varname$superscript\_subscript$@[style]

where pre-scripts are surrounded by percent signs, and post-scripts are surrounded by dollar signs. If there are no pre-scripts, the percent signs can be omitted, and if there are no post-scripts, the dollar signs can be omitted. [style] is a sequence of characters to indicate font and style, and can include:

* U to display the variable name in a "Upright" rather than "Italic" posture (default is Italic)
* At most one of D, S, C to choose “Double-Struck”, “Script”, or “Calligraphic” font (default is Roman)
* An optional B to choose bold-face (default is no boldface)
* At most one of v, d, dd, p, pp, ppp, h, w, t, o, oo, u, uu to add “vector”, “dot”, “double dot”, “prime”, “double prime”, “triple prime”, “hat”, “wide hat”, “tilde”, “overline” “double overline”, “underline”, or “double underline” accent (default is no accent, if multiple tags are found, only one will be used, and which gets used is implementation-dependent)

The variable name, superscripts, and subscripts can include backslash-escapes to access non-standard characters like Greek letters, etc.

**Data Types**

The set of supported data types are shown below, with indentation representing subclass relationships (for example, an **Integer** is also a **Real Number** is also a **Complex Number**).

* **Complex Number** (**abstract**)
  + **Complex Cartesian** (two real numbers representing real and imaginary parts)
  + **Complex Polar** (two real numbers representing radius and polar angle)
  + **Real Number** (**abstract**)
    - **Floating Point** (a double-precision floating point, including and but excluding NaN)
    - **Big Decimal** (an arbitrary precision decimal value)
    - **Rational** (a **Long** numerator and denominator, always in reduced form)
    - **Irrational** (a “common” irrational, represented by a structure like a rational but in which the numerator is multiplied by , , or a square root of a 31-bit positive integer)
    - **Integer** (**abstract**)
      * **Long** (a 64-bit signed integer)
      * **Big Integer** (an integer of arbitrary size)
* **Complex Matrix**
  + **Complex Vector** (a one-column matrix)
  + **Real Matrix**
    - **Real Vector** (a one-column matrix)
    - **Integer Matrix**
      * **Integer Vector** (a one-column matrix)
* **Point 3D** (**abstract**; includes flag to transform as a vector)
  + **Point 3D Cartesian** (real finite coordinates)
  + **Point 3D Cylindrical** (real finite coordinates)
  + **Point 3D Spherical** (real finite coordinates)
  + **Point 2D (abstract)** (treated as having )
    - **Point 2D Cartesian** (real finite coordinates)
    - **Point 2D Polar** (real finite coordinates)
* **Set** (**abstract**)
  + **Number Set** (**abstract**)
    - **Complex Set** (only finite values permitted)
      * **Real Set**
        + **Integer Set**
  + **Point Set** (**abstract**)
    - **Point 3D Set**
      * **Point 2D Set**
* **Interval** (real upper and lower bounds, closure flag for each bound, and an “inverted” flag)
* **Boolean**
* **String** (a sequence of Unicode code points)
* **Error** (indicates an evaluation error, like division by zero, or inverse sine of )

**Visual Representation and Editing**

Every expression node can have a visual representation attached that can support rendering of the node and editing. The visual representation will include bounding boxes for all tokens (where a “token” is an atomic unit that the cursor can move over in one step, and that is either selected or unselected when a selection range is active).

The bounding boxes are stored separately from the actual rendered implementation and are updated based on the actual rendering implementations’ calculations of sizes and positioning. The bounding box structure is then used to process cursor movements, mouse click/drag operations, cut and paste, insertion, and so forth. The rendering implementation is simply responsible for drawing the content and updating size and position information as needed.

When an expression has visual rendering information attached, it will also have a Rendering Context object that stores several parameters that can control rendering:

* Base font size (size is adjusted for subscripts, superscripts, and “dense” constructions like fractions).
* Minimum font size
* Presentation vs. inline
* Script size multiplier (default is 71%)
* Fraction size multiplier (default is 85%)
* Line thickness (for fraction bars, top lines of radicals, etc.)
* Math color (for function names, operators, fences, commas, ellipses, etc.)
* Decoration color (for fraction lines, radicals)
* Variable color (for variable references)
* Quantity color (for numbers, vectors, matrices)
* Background color
* Cursor color
* Selection color
* Gap width
* Gap dash color
* Gap dash pattern
* Operator spacing (space to apply on each side of an operator)
* Function argument spacing (space between arguments and parentheses)
* List spacing (space between comma and next item in a list or vector)
* Matrix column gap
* Matrix row gap
* Matrix brackets inner padding
* Matrix brackets outer spacing

**Structural Representation**

There are structural tree node types to represent a constant value of each of these data types except **Error**, which can only be generated through evaluation.Other structural nodes are described below.

**Variable Reference**

A reference to the value of a variable in the evaluation context, consisting of a name and zero, one, or two positive integer indices, like “a”, “a[2]”, or “a[2, 1]”. This type of tree node has no child nodes.

With no indices the reference evaluates to the corresponding variable value from the evaluation context at the time of evaluation, or an **Error** if there is no variable in the context with the specified name.

With a single index, if the value is a vector, the result is the numeric entry specified by the index (where [1] indicates the first entry). If the index is greater than the number of entries in the vector, an **Error** is generated. For any other value type, attempting to use an index will generate an **Error**.

With two indices, if the value is a matrix, the numeric entry is returned at the row of the first index, and the column of the second index (where [2, 1] indicates the second row and first column). If the first index is greater than the number of rows or the second is greater than the number of columns, an **Error** is generated. For any other value type, attempting to use an index will generate an **Error**.

***Visual Layout***

The tokens in a variable name include (in this order):

• If there is a presuperscript or presubscript, an invisible token where the cursor steps into the first  
 of these;

• If there is a presuperscript, the presuperscript characters and an invisible token where the cursor steps  
 from the end of the presuperscript into the next block;

• If there is a presubscript, the presubscript characters and an invisible token where the cursor steps  
 from the end of the presubscript into the next block;

• The variable name characters;

• If there is a superscript, an invisible token where the cursor steps into the superscript, then the  
 superscript characters;

• If there is a subscript, an invisible token where the cursor steps into the subscript, then the, the  
 subscript characters;

• If there is a superscript or subscript, an invisible character where the cursor steps from the last script  
 to the position after the complete variable identifier;

• If there is an index:

• A ‘[‘ character;

• The digits of the first index;

• If there is a second index:

• A ‘,‘ character with a thin space;

• The digits of the second index;

• A ‘]‘ character.

There will be layout boxes, each with (x, y) position, width, height, ascent, and descent, for the presuperscript, presubscript, variable name, accent on the variable name, superscript, subscript, opening index bracket, first index, comma/space, second index, and closing index bracket. The entire construction will have its own layout box.

Variable references are laid out on the text baseline of the line in which they appear. Any accents on the variable name are included in the layout bounds of the name, but the baseline for superscripts is the same as for names with no accent.

**Unary Function**

A function or operator with a single argument. A tree node for a unary function specifies an “operation” (see below) and has a single child node whose value is computed and then the function is applied to that value. Some unary operators support “parameters”, like the base for a logarithm or the variable name for differentiation.

Unary functions have varying visual representations – a unary “negation” operator is represented with a simple leading negation sign, a “sine” function is represented in function notation, and other functions like differentiation and integration have their own special visual representations.

Some unary functions have parameters, like the differentiation variable, integration limits, angle units, etc. The defined unary functions, and their operation on the various data types are described below.

We list the unary functions by the type of argument – many functions will appear under several different argument types, but this allows for more coherent descriptions of function operations.

**function (Integer )**

**Factorial**

If is non-negative, evaluates to or to an **Error** if the result is too large.

**Radian Numerator**

For degrees, evaluates to the numerator of the rational multiple of that represents the angle in radians. For example, if , where the radian measure of degrees is , this function evaluates to 2.

**Radian Denominator**

For degrees, evaluates to the denominator of the rational multiple of that represents the angle in radians. For example, if , where the radian measure of degrees is , this function evaluates to .

**Greatest Powered Factor** [Parameter: **Power** ( or greater, default is )]

Evaluates to the greatest integer that, when raised to some integer power, is a divisor of . If **Power** is , this evaluates to the greatest integer whose square divides . If **Power** is , this evaluates to the greatest integer whose cube divides . This is useful for simplifying square roots, cube roots, etc. For example, if and **Power** is , this evaluates to since is the greatest perfect square that divides . If the and **Power** is , this evaluates to since is the greatest perfect cube that divides .

**Identity Matrix**

Evaluates to an identity matrix.

**Permute**

Evaluates to an integer vector with the numbers to randomly permuted, or to an **Error** if is less than .

**function (Real Number )**

**Absolute Value**

Evaluates to if is non-negative or otherwise. To find the magnitude of a complex number, use **Norm**.

**Floor**

Evaluates to the greatest **Integer** less than or equal to (or an **Error** if the input is infinite).

**Ceiling**

Evaluates to the least **Integer** greater than or equal to (or an **Error** if the input is infinite).

**Round To Integer** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]

Evaluates to the nearest **Integer** to (or an **Error** if is infinite), treating numbers halfway between two integers according to its **Mode** setting.

**Round** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]  
 [Parameter: **Digits** (non-negative integer number of digits after radix, default is 0)]

Evaluates to the result of rounding to the nearest value with no more than the specified number of digits after the radix (or an **Error** if the input is infinite).

**Exponential** [Parameter: **Base** (a positive real number, default is )]

Evaluates to .

**Logarithm** [Parameter: **Base** (a positive real number, default is )]

Evaluates to the logarithm the with the specified base.

**Signum**

Evaluates to , , or as is negative, zero, or positive.

**Radians To Degrees**

Evaluates the degree measure corresponding to treated as a radian measure (the input times then divided by )

**Degrees To Radians**

Evaluates the radian measure corresponding to treated as a degree measure (the input times then divided by )

**function (Complex Number )**

**Negation**

Evaluates to the negation of (for complex polar numbers, this will adjust the angle by half a turn rather than making the radius negative).

**Complex Conjugate**

Evaluates to the complex conjugate of .

**Real Part**

Evaluates to the real part of .

**Imaginary Part**

Evaluates to the imaginary part of .

**Complex Argument**

Evaluates to the argument (polar angle) of .

**Norm**

Evaluates to the magnitude of . For real values, this agrees with **Absolute Value**.

**Exponential** [Parameter: **Base** (a positive real number, default is )]

Evaluates to .

**Logarithm** [Parameter: **Base** (a positive real number, default is )]

Evaluates to the logarithm the with the specified base (the principal branch).

**Sin, Cosine, Tangent, Secant, Cosecant, Cotangent, Inverse Sine, Inverse Cosine, Inverse Tangent, Inverse Secant, Inverse Cosecant, Inverse Cotangent, Hyperbolic Sine, Hyperbolic Cosine, Hyperbolic Tangent, Hyperbolic Secant, Hyperbolic Cosecant, Hyperbolic Cotangent, Inverse Hyperbolic Sine, Inverse Hyperbolic Cosine, Inverse Hyperbolic Tangent, Inverse Hyperbolic Secant, Inverse Hyperbolic Cosecant, Inverse Hyperbolic Cotangent** [Parameter: **Units** (in {degrees, radians}, default is degrees)]

Evaluates to the appropriate trigonometric function of (**Error** if the input is outside the domain).

**Root** [Parameter: **N** (an integer or greater, default is )]

Evaluates to the -th root of , or **Error** if is negative, the root is even, and the evaluation context does not allow complex values.

**function (Integer Vector )**

**Least Common Multiple**

Evaluates to the integer least common multiple of the entries in .

**Greatest Common Divisor**

Evaluates to the integer greatest common divisor of the entries in .

**function (Real Matrix )**

**Absolute Value**

Evaluates to a matrix the same size as where all constituent numeric values have positive sign.

**Floor**

Evaluates to an integer matrix the same size as where each entry is the floor of the entry in .

**Ceiling**

Evaluates to an integer matrix the same size as where each entry is the ceiling of the entry in .

**Round To Integer** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]

Evaluates to an integer matrix the same size as where each entry is the nearest integer to the corresponding entry in .

**Round** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]  
 [Parameter: **Digits** (non-negative integer number of digits after radix, default is 0)]

Evaluates to a matrix the same size as where each entry is the corresponding entry in the input rounded to the specified number of digits after the radix.

**Least**

Evaluates to the least entry in .

**Greatest**

Evaluates to the greatest entry in .

**Median**

Evaluates to the median of all entries in .

**Variance**

Evaluates to the variance of all entries in .

**Standard Deviation**

Evaluates to the standard deviation of all entries in .

**Containing Interval**

Evaluates to the smallest **Interval** that contains all entries in .

**function (Complex Matrix )**

**Number of Rows**

Evaluates to the positive integer number of rows in .

**Number of Columns**

Evaluates to the positive integer number of columns in .

**Negation**

Evaluates to a matrix the same size as where all constituent numeric values have opposite sign.

**Complex Conjugate**

Evaluates to a matrix the same size as where each constituent complex value has been conjugated.

**Real Part**

Evaluates to a real matrix the same size as where each entry is the real part of the corresponding entry in .

**Imaginary Part**

Evaluates to a real matrix the same size as where each entry is the imaginary part of the corresponding entry in .

**Complex Argument**

Evaluates to a real matrix the same size as where each entry is the argument (polar angle) of the corresponding entry in .

**Entry Norm**

Evaluates to a real matrix the same size as where each entry is the norm of the corresponding entry in .

**Vector Norm**

Evaluates to the 2-norm of a (for matrices, this is the 2-norm induced by the vector 2-norm).

**Mean**

Evaluates to the mean or all entries in .

**Determinant**

If is square, evaluates to the determinant; otherwise evaluates to **Error**.

**Transpose**

Evaluates to the transpose of , where the transpose of a one-row matrix will be returned as a vector.

**Trace**

If is square, evaluates to the trace; otherwise evaluates to **Error**.

**Matrix Inverse**

If is square and invertible, evaluates to the inverse; otherwise evaluates to **Error**.

**To Set**

Evaluates to a set containing all distinct entries in .

**function (Point 3D )**

**Negation**

Evaluates to reflected across the origin.

**Absolute Value**

Evaluates to a point whose Cartesian coordinates are the absolute value of the Cartesian coordinates of .

**Norm**

Evaluates the distance from the origin to .

**X Coordinate**

Evaluates to the coordinate of .

**Y Coordinate**

Evaluates to the coordinate of .

**Z Coordinate**

Evaluates to the coordinate of (zero for 2-D points).

**R Coordinate**

Evaluates to the coordinate of (equivalent to **Norm**).

**Theta Coordinate** [Parameter: **Units** (in {degrees, radians}, default is degrees)]

Evaluates to the (azimuthal angle) coordinate of the point, zero on the axis.

**Phi Coordinate** [Parameter: **Units** (in {degrees, radians}, default is degrees)]

Evaluates to the (polar angle) coordinate of the point, zero at the pole.

**Normalize**

Evaluates to a point whose coordinates are the coordinates of divided by its norm, or **Error** if lies at the origin.

**To Vector**

Evaluates to a real vector with 2 or 3 components set to the Cartesian coordinates of the point.

**function (Integer Set )**

**Least Common Multiple**

Evaluates to the least common multiple of the entries in .

**Greatest Common Divisor**

Evaluates to the greatest common divisor of the entries in .

**function (Real Set )**

**Absolute Value**

Evaluates to a set whose values are the absolute value of the values in (the result may be a smaller set than ).

**Floor**

Evaluates to an integer set of the same size as where each entry is the floor of the entry in (the result may be a smaller set than ).

**Ceiling**

Evaluates to an integer set of the same size as where each entry is the ceiling of the entry in (the result may be a smaller set than ).

**Round To Integer** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]

Evaluates to an integer set where each entry is nearest integer to the entry in (the result may be a smaller set than ).

**Round** [Parameter: **Mode** (in {half-up, half-down}, default is half-up)]  
 [Parameter: **Digits** (non-negative integer number of digits after radix, default is 0)]

Evaluates to a real set where each entry is the corresponding entry in rounded to the specified number of digits after the radix (the result may be a smaller set than ).

**Least**

Evaluates to the least entry in .

**Greatest**

Evaluates to the greatest entry in .

**Median**

Evaluates to the median of all entries in .

**Variance**

Evaluates to the variance of all entries in .

**Standard Deviation**

Evaluates to the standard deviation of all entries in .

**Containing Interval**

Evaluates to the smallest **Interval** that contains all entries in .

**function (Number Set )**

**Complex Conjugate**

Evaluates to a set whose values are the complex conjugates of the values in .

**function (Set )**

**Negation**

Evaluates to a set whose values are the negations of the values in .

**Cardinality**

Evaluates to the non-negative integer cardinality of .

**Permute**

Generates a vector with the entries from permuted randomly.

**Mean**

Evaluates to the mean of all entries in , where the mean of a set of points is a point whose Cartesian coordinates are the mean of the corresponding Cartesian coordinate of all points in the set.

**function (Interval )**

**Negation**

Evaluates to an interval whose bounds are the negation of ’s bounds.

**Logical Not**

Evaluates to an interval with the same bounds as but the opposite “inverted” flag.

**Norm**

Evaluates to the non-negative length of ( if unbounded)

**Floor**

Evaluates to integer floor of the lower bound of (or an **Error** if unbounded)

**Ceiling**

Evaluates to integer ceiling of the upper bound of (or an **Error** if unbounded)

**Least**

Evaluates to the lower bound of (if not bounded below).

**Greatest**

Evaluates to the upper bound of (if not bounded above).

**Mean**

Evaluates to the average of the lower and upper bound of (or an **Error** if unbounded)

**function (Boolean )**

**Logical Not**

Evaluates to the opposite **Boolean** value.

**function (String )**

**Length**

Evaluates to the non-negative integer length of , in code points.

**To Uppercase**

Evaluates to a **String** of the same length with each code point of converted to uppercase.

**To Lowercase**

Evaluates to a **String** of the same length with each code point of converted to lowercase.

**Permute**

Evaluates to a **String** with the letters of randomly permuted.

**Binary Function**

A function with two arguments. A tree node for a binary function specifies an “operation” (see below) and has an ordered list of two child nodes whose values are computed and then the function is applied to those value. Some binary operators support “parameters”.

Binary functions have varying visual representations – some are presented as inline operations like subtraction, or as fractions for division, or as a function invocation.

The defined binary functions, and their operation on the various data types are described below.

We list the binary functions by the type of argument – many functions will appear under several different argument types, but this allows for more coherent descriptions of function operations.

**function (Integer , Integer )**

**Integer Remainder**

Evaluates to the remainder after dividing by . The result is negative if exactly one of and are negative, positive otherwise.

**Integer Quotient**

Evaluates to the quotient after dividing by . The quotient is the greatest whole number of that can go into , negative if exactly one of and are negative, positive otherwise.

**Integer Modulo**

Evaluates to modulo if is a positive integer, **Error** if not. The result will be an integer between 0 (inclusive) and (exclusive).

**Divides**

Evaluates to TRUE if divides ; FALSE if not, **Error** if or is zero.

**Combinations**

Evaluates to the binomial coefficient , which is the number of combinations of items chosen from a set of items ( choose ).

**Permutations**

Evaluates to the number of permutations that can be formed when choosing items from a set of items.

**function (Real Number , Real Number )**

**Hypotenuse**

Evaluates to .

**Less Than**

Evaluates to TRUE, if , FALSE otherwise.

**Less Than or Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Greater Than**

Evaluates to TRUE, if , FALSE otherwise.

**Greater Than or Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Approximately Equal To** [Parameter: **Tolerance** (a positive real number )]

Evaluates to TRUE, if , FALSE otherwise.

**To Point 2D**

Evaluates to a new point with coordinates ; **Error** if either coordinate is infinite.

**function (Complex Number , Complex Number )**

**Subtraction**

Evaluates to .

**Division**

Evaluates to , or **Error** if .

**Exponentiation**

Evaluates to , or **Error** if or if is negative and is not an integer.

**Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Not Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**function (Complex Matrix , Complex Matrix )**

**Subtraction**

Evaluates to the if those matrices are the same size, or **Error** if not.

**function (Complex Vector , Complex Vector )**

**Inner Product**

Evaluates to the complex inner product of and if those vectors are the same size, or **Error** if not.

**function (Complex Matrix , Complex Number )**

**Division**

Evaluates to a matrix the size of whose entries are the entries of divided by if those matrices are the same size, or **Error** if not.

**function (Complex Matrix , Complex Number )**

**Division**

Evaluates to a matrix the size of whose entries are the entries of divided by if those matrices are the same size, or **Error** if not.

**function (Point 2D , Point 2D )**

**Subtraction**

Evaluates to a **Point 2D** with coordinates .

**function (Point 3D , Point 3D )**

**Subtraction**

Evaluates to a **Point 3D** with coordinates .

**Inner Product**

Evaluates to the inner product of and .

**Cross Product**

Evaluates to a point whose coordinates are .

**function (Point 3D , Real Number )**

**Multiplication**

Evaluates to a **Point 3D** of the same type as scaled by factor .

**function (Complex Number , Number Set )**

**Element Of**

Evaluates to TRUE if is an element in ; FALSE if not.

**Not Element Of**

Evaluates to FALSE if is an element in ; TRUE if not.

**function (Point 3D , Point Set )**

**Element Of**

Evaluates to TRUE if is an element in ; FALSE if not.

**Not Element Of**

Evaluates to FALSE if is an element in ; TRUE if not.

**function (Number Set , Number Set )**

**Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Not Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Subset Of**

Evaluates to TRUE if is a subset of ; FALSE if not.

**Not Subset Of**

Evaluates to FALSE if is a subset of ; TRUE if not.

**Set Difference**

Evaluates to a set with all entries in that are not in .

**function (Point Set , Point Set )**

**Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Not Equal To**

Evaluates to TRUE, if , FALSE otherwise.

**Subset Of**

Evaluates to TRUE if is a subset of ; FALSE if not.

**Not Subset Of**

Evaluates to FALSE if is a subset of ; TRUE if not.

**Set Difference**

Evaluates to a set with all entries in that are not in .

**function (Boolean , Boolean )**

**Exclusive Or**

Evaluates to TRUE, if , FALSE otherwise.

**Trinary Function**

A function with three arguments. A tree node for a trinary function specifies an “operation” (see below) and has an ordered list of three child nodes whose values are computed and then the function is applied to those value.

The defined trinary functions, and their operation on the various data types are described below.

We list the trinary functions by the type of argument – many functions will appear under several different argument types, but this allows for more coherent descriptions of function operations.

**function (Integer , Integer , Integer )**

**Sum**

Evaluates the sum of integers from to (inclusive), with step size ; **Error** if is non-positive. For example, if , , , the result is .

**Product**

Evaluates the product of integers from to (inclusive), with step size ; **Error** if is non-positive. For example, if , , , the result is .

**function (Real Number , Real Number , Real Number )**

**To Point 3D**

Evaluates to a new point with coordinates ; **Error** if any coordinate is infinite.

**N-ary Function**

A function with one or more arguments. A tree node for a n-ary function specifies an “operation” (see below) and has an ordered list of child nodes whose values are computed and then the function is applied to those value.

The defined n-ary functions, and their operation on the various data types are described below.

We list the n-ary functions by the type of argument – many functions will appear under several different argument types, but this allows for more coherent descriptions of function operations.

**function (Integer , Integer**   **Integer )**

**Least Common Multiple**

Evaluates to the least common multiple of the entries in .

**Greatest Common Divisor**

Evaluates to the greatest common divisor of the entries in .

**function (Real Number , Real Number**   **Real Number )**

**Least**

Evaluates to the least entry in .

**Greatest**

Evaluates to the greatest entry in .

**Median**

Evaluates to the median of all entries in .

**Variance**

Evaluates to the variance of all entries in .

**Standard Deviation**

Evaluates to the standard deviation of all entries in .

**Containing Interval**

Evaluates to the smallest **Interval** that contains all entries in .

**Permute**

Generates a vector with the entries from permuted randomly.

**Is Increasing** [Parameter: **Strict** (Boolean, default is FALSE)]

Returns TRUE if the sequence is (strictly) increasing, FALSE otherwise.

**Is Decreasing** [Parameter: **Strict** (Boolean, default is FALSE)]

Returns TRUE if the sequence is (strictly) decreasing, FALSE otherwise.

**function (Complex Number , Complex Number**   **Complex Number )**

**Addition**

Evaluates to the ; **Error** if inputs contain both and or any real infinity and any complex numbers with nonzero imaginary part.

**Multiplication**

Evaluates to the ; **Error** if inputs contain any real infinity and any complex numbers with nonzero imaginary part.

**To Vector**

Evaluates to the vector with entries .

**To Set**

Evaluates to the set with entries .

**Mean**

Evaluates to the mean of all entries in .

**Equals**

Returns TRUE if , FALSE otherwise.

**function (Integer , Integer , Complex Number , Complex Number**   **Complex Number )**

**To Matrix**

Evaluates to a matrix with the specified number of columns and rows, and with entries , where any missing entries are filled with zeros; **Error** if too many entries are provided.

**function (Complex Matrix , Complex Matrix**   **Complex Matrix )**

**Addition**

Evaluates to the ; **Error** if all inputs are not the same size.

**function (Number Set , Number Set**   **Number Set )**

**Union**

Evaluates to the set union of all input sets.

**Intersection**

Evaluates to the set intersection of all input sets.

**Is Subset** [Parameter: **Proper** (Boolean, default is FALSE)]

Returns TRUE if is a (proper) subset of , etc.; FALSE otherwise.

**Is Superset** [Parameter: **Proper** (Boolean, default is FALSE)]

Returns TRUE if is a (proper) superset of , etc.; FALSE otherwise.

**function (Point Set , Point Set**   **Point Set )**

**Union**

Evaluates to the set union of all input sets.

**Intersection**

Evaluates to the set intersection of all input sets.

**Is Subset** [Parameter: **Proper** (Boolean, default is FALSE)]

Returns TRUE if is a (proper) subset of , etc.; FALSE otherwise.

**Is Superset** [Parameter: **Proper** (Boolean, default is FALSE)]

Returns TRUE if is a (proper) superset of , etc.; FALSE otherwise.

**function (Point 3D , Point 3D**   **Point 3D )**

**To Set**

Evaluates to the set of all input points

**function (Boolean , Boolean**   **Boolean )**

**And**

Evaluates to the .

**Or**

Evaluates to the .

**Functions using Expressions**

Some functions use expressions that are evaluated at the time the function is evaluated.

For example, conditional functions select which result to return based on a test condition, which is itself an expression that can be evaluated. These functions support any type of result value, but all results must be of compatible type so the result of evaluating the conditional function is well-defined.

**function (Expression , Any , Any )**

**If Then Else**

Evaluates to if evaluates to TRUE, if evaluates to FALSE; **Error** if evaluates to **Error** or something other than a **Boolean**.

**function (Expression , Integer , Any**  **Integer , Any )**

**Switch**

Evaluates to if evaluates to , if evaluates to ; **Error** if evaluates to **Error** or something other than an **Integer**.

**function (String , Interval , Expression**  **Interval , Expression )**

**Piecewise**

Evaluates to if has a value in , if has a value in ; **Error** if has no value or its value does not fall in any case. Cases should be disjoint, but if not, the first matching case is used.

**function (Expression , String )**

**Derivative**

Evaluates to a new expression that is the derivative of with respect to ; **Error** if cannot be differentiated.

**Antiderivative**

Evaluates to a new expression that is the antiderivative of with respect to ; **Error** if cannot be antidifferentiated.

**function (Expression , String , Real Number , Real Number )**

**Integral**

Evaluates the definite integral of with respect to over the given bounds; **Error** if the integral cannot be found.