Expression syntax

This page describes the syntax of expression parser of math.js. It describes how to work with the available data types, functions, operators, variables, and more.

Differences from JavaScript

The expression parser of math.js is aimed at a mathematical audience, not a programming audience. The syntax is similar to most calculators and mathematical applications. This is close to JavaScript as well, though there are a few important differences between the syntax of the expression parser and the lower level syntax of math.js. Differences are:

- No need to prefix functions and constants with the `math.*` namespace, you can just enter `sin(pi / 4)`.
- Matrix indexes are one-based instead of zero-based.
- There are index and range operators which allow more conveniently getting and setting matrix indexes, like `A[2:4, 1]`.
- Both indexes and ranges and have the upper-bound included.
- There is a differing syntax for defining functions. Example: `f(x) = x^2`.
- There are custom operators like `x + y` instead of `add(x, y)`.
- Some operators are different. For example `^` is used for exponentiation, not bitwise xor.
- Implicit multiplication, like `2 pi`, is supported and has special rules.
- Relational operators (`<`, `>`, `<=`, `>=`, `==`, and `!=`) are chained, so the expression `5 < x < 10` is equivalent to `5 < x and x < 10`.

Operators

The expression parser has operators for all common arithmetic operations such as addition and multiplication. The expression parser uses conventional infix notation for operators: an operator is placed between its arguments. Round parentheses can be used to override the default precedence of operators.

```
// use operators
math.eval('2 + 3')       // 5
math.eval('2 * 3')       // 6
```

```
// use parentheses to override the default precedence
math.eval('2 + 3 * 4')   // 14
math.eval('(2 + 3) * 4') // 20
```

The following operators are available:

Operator	Name	Syntax	Associativity	Example	Result
`( )`	Grouping	`(x)`	None	`2 * (3 + 4)`	`14`
`[ ]`	Matrix, Index	`[...]`	None	`[[1,2],[3,4]]`	`[[1,2], [3,4]]`
`{ }`	Object	`{...}`	None	`{a: 1, b: 2}`	`{a: 1, b: 2}`
`,`	Parameter separator	`x, y`	Left to right	`max(2, 1, 5)`	`5`
`.`	Property accessor	`obj.prop`	Left to right	`obj={a: 12}; obj.a`	`12`
`;`	Statement separator	`x; y`	Left to right	`a=2; b=3; a*b`	`[6]`
`;`	Row separator	`[x; y]`	Left to right	`[1,2;3,4]`	`[[1,2], [3,4]]`
`\n`	Statement separator	`x \n y`	Left to right	`a=2 \n b=3 \n a*b`	`[2,3,6]`
`+`	Add	`x + y`	Left to right	`4 + 5`	`9`
`+`	Unary plus	`+y`	Right to left	`+4`	`4`
`-`	Subtract	`x - y`	Left to right	`7 - 3`	`4`
`-`	Unary minus	`-y`	Right to left	`-4`	`-4`
`*`	Multiply	`x * y`	Left to right	`2 * 3`	`6`
`.*`	Element-wise multiply	`x .* y`	Left to right	`[1,2,3] .* [1,2,3]`	`[1,4,9]`
`/`	Divide	`x / y`	Left to right	`6 / 2`	`3`
`./`	Element-wise divide	`x ./ y`	Left to right	`[9,6,4] ./ [3,2,2]`	`[3,3,2]`
`%, mod`	Modulus	`x % y`	Left to right	`8 % 3`	`2`
`^`	Power	`x ^ y`	Right to left	`2 ^ 3`	`8`

Operator	Name	Syntax	Associativity	Example	Result
`.^`	Element-wise power	`x .^ y`	Right to left	`[2,3] .^ [3,3]`	`[9,27]`
`'`	Transpose	`y'`	Left to right	`[[1,2],[3,4]]'`	`[[1,3], [2,4]]`
`!`	Factorial	`y!`	Left to right	`5!`	`120`
`&`	Bitwise and	`x & y`	Left to right	`5 & 3`	`1`
`~`	Bitwise not	`~x`	Right to left	`~2`	`-3`
`|`	Bitwise or	`x | y`	Left to right	`5 | 3`	`7`
`^|`	Bitwise xor	`x ^| y`	Left to right	`5 ^| 2`	`7`
`<<`	Left shift	`x << y`	Left to right	`4 << 1`	`8`
`>>`	Right arithmetic shift	`x >> y`	Left to right	`8 >> 1`	`4`
`>>>`	Right logical shift	`x >>> y`	Left to right	`-8 >>> 1`	`2147483644`
`and`	Logical and	`x and y`	Left to right	`true and false`	`false`
`not`	Logical not	`not y`	Right to left	`not true`	`false`
`or`	Logical or	`x or y`	Left to right	`true or false`	`true`
`xor`	Logical xor	`x xor y`	Left to right	`true xor true`	`false`
`=`	Assignment	`x = y`	Right to left	`a = 5`	`5`
`? :`	Conditional expression	`x ? y : z`	Right to left	`15 > 100 ? 1 : -1`	`-1`
`:`	Range	`x : y`	Right to left	`1:4`	`[1,2,3,4]`
`to, in`	Unit conversion	`x to y`	Left to right	`2 inch to cm`	`5.08 cm`
`==`	Equal	`x == y`	Left to right	`2 == 4 - 2`	`true`
`!=`	Unequal	`x != y`	Left to right	`2 != 3`	`true`
`<`	Smaller	`x < y`	Left to right	`2 < 3`	`true`
`>`	Larger	`x > y`	Left to right	`2 > 3`	`false`
`<=`	Smallereq	`x <= y`	Left to right	`4 <= 3`	`false`
`>=`	Largereq	`x >= y`	Left to right	`2 + 4 >= 6`	`true`

Precedence

The operators have the following precedence, from highest to lowest:

Operators	Description

`(...)`
`[...]`
`{...}`
	Grouping
	Matrix
	Object

`x(...)`
`x[...]`
`obj.prop`
`:`
	Function call
	Matrix index
	Property accessor
	Key/value separator

`'`	Matrix transpose
`!`	Factorial
`^, .^`	Exponentiation
`+, -, ~, not`	Unary plus, unary minus, bitwise not, logical not
See section below	Implicit multiplication
`*, /, .*, ./, %, mod`	Multiply, divide, modulus
`+, -`	Add, subtract
`:`	Range
`to, in`	Unit conversion
`<<, >>, >>>`	Bitwise left shift, bitwise right arithmetic shift, bitwise right logical shift
`==, !=, <, >, <=, >=`	Relational
`&`	Bitwise and
`^|`	Bitwise xor
`|`	Bitwise or
`and`	Logical and
`xor`	Logical xor
`or`	Logical or
`?, :`	Conditional expression
`=`	Assignment
`,`	Parameter and column separator
`;`	Row separator
`\n, ;`	Statement separators

Functions

Functions are called by entering their name, followed by zero or more arguments enclosed by parentheses. All available functions are listed on the page Functions.

```
math.eval('sqrt(25)')           // 5
math.eval('log(10000, 3 + 7)')  // 4
math.eval('sin(pi / 4)')        // 0.7071067811865475
```

New functions can be defined using the `function` keyword. Functions can be defined with multiple variables. Function assignments are limited: they can only be defined on a single line.

```
const parser = math.parser()

parser.eval('f(x) = x ^ 2 - 5')
parser.eval('f(2)')     // -1
parser.eval('f(3)')     // 4

parser.eval('g(x, y) = x ^ y')
parser.eval('g(2, 3)')  // 8
```

Math.js itself heavily uses typed functions, which ensure correct inputs and throws meaningful errors when the input arguments are invalid. One can create a typed-function in the expression parser like:

```
const parser = math.parser()

parser.eval('f = typed({"number": f(x) = x ^ 2 - 5})')
```

Constants and variables

Math.js has a number of built-in constants such as `pi` and `e`. All available constants are listed on he page Constants.

```
// use constants
math.eval('pi')                 // 3.141592653589793
math.eval('e ^ 2')              // 7.3890560989306495
math.eval('log(e)')             // 1
math.eval('e ^ (pi * i) + 1')   // ~0 (Euler)
```

Variables can be defined using the assignment operator `=`, and can be used like constants.

```
const parser = math.parser()

// define variables
parser.eval('a = 3.4')      // 3.4
parser.eval('b = 5 / 2')    // 2.5

// use variables
parser.eval('a * b')        // 8.5
```

Variable names must:

- Begin with an “alpha character”, which is:

    - A latin letter (upper or lower case). Ascii: `a-z, A-Z`
    - An underscore. Ascii: `_`
    - A dollar sign. Ascii: `$`
    - A latin letter with accents. Unicode: `\u00C0 - \u02AF`
    - A greek letter. Unicode: `\u0370 - \u03FF`
    - A letter-like character. Unicode: `\u2100 - \u214F`
    - A mathematical alphanumeric symbol. Unicode: `\u{1D400} - \u{1D7FF}` excluding invalid code points
- Contain only alpha characters (above) and digits `0-9`
- Not be any of the following: `mod, to, in, and, xor, or, not, end`. It is possible to assign to some of these, but that's not recommended.

It is possible to customize the allowed alpha characters, see Customize supported characters for more information.

Data types

The expression parser supports booleans, numbers, complex numbers, units, strings, matrices, and objects.

Booleans

Booleans `true` and `false` can be used in expressions.

```
// use booleans
math.eval('true')               // true
math.eval('false')              // false
math.eval('(2 == 3) == false')  // true
```

Booleans can be converted to numbers and strings and vice versa using the functions `number` and `boolean`, and `string`.

```
// convert booleans
math.eval('number(true)')       // 1
math.eval('string(false)')      // "false"
math.eval('boolean(1)')         // true
math.eval('boolean("false")')   // false
```

Numbers

The most important and basic data type in math.js are numbers. Numbers use a point as decimal mark. Numbers can be entered with exponential notation. Examples:

```
// numbers in math.js
math.eval('2')        // 2
math.eval('3.14')     // 3.14
math.eval('1.4e3')    // 1400
math.eval('22e-3')    // 0.022
```

A number can be converted to a string and vice versa using the functions `number` and `string`.

```
// convert a string into a number
math.eval('number("2.3")')    // 2.3
math.eval('string(2.3)')      // "2.3"
```

Math.js uses regular JavaScript numbers, which are floating points with a limited precision and limited range. The limitations are described in detail on the page Numbers.

```
math.eval('1e-325')   // 0
math.eval('1e309')    // Infinity
math.eval('-1e309')   // -Infinity
```

When doing calculations with floats, one can very easily get round-off errors:

```
// round-off error due to limited floating point precision
math.eval('0.1 + 0.2')  // 0.30000000000000004
```

When outputting results, the function `math.format` can be used to hide these round-off errors when outputting results for the user:

```
const ans = math.eval('0.1 + 0.2')  //  0.30000000000000004
math.format(ans, {precision: 14})   // "0.3"
```

BigNumbers

Math.js supports BigNumbers for calculations with an arbitrary precision. The pros and cons of Number and BigNumber are explained in detail on the page Numbers.

BigNumbers are slower but have a higher precision. Calculations with big numbers are supported only by arithmetic functions.

BigNumbers can be created using the `bignumber` function:

```
math.eval('bignumber(0.1) + bignumber(0.2)')  // BigNumber, 0.3
```

The default number type of the expression parser can be changed at instantiation of math.js. The expression parser parses numbers as BigNumber by default:

```
// Configure the type of number: 'number' (default), 'BigNumber', or 'Fraction'
math.config({number: 'BigNumber'})

// all numbers are parsed as BigNumber
math.eval('0.1 + 0.2')  // BigNumber, 0.3
```

BigNumbers can be converted to numbers and vice versa using the functions `number` and `bignumber`. When converting a BigNumber to a Number, the high precision of the BigNumber will be lost. When a BigNumber is too large to be represented as Number, it will be initialized as `Infinity`.

Complex numbers

Complex numbers can be created using the imaginary unit `i`, which is defined as `i^2 = -1`. Complex numbers have a real and complex part, which can be retrieved using the functions `re` and `im`.

```
const parser = math.parser()

// create complex numbers
parser.eval('a = 2 + 3i')   // Complex, 2 + 3i
parser.eval('b = 4 - i')    // Complex, 4 - i

// get real and imaginary part of a complex number
parser.eval('re(a)')        // Number,  2
parser.eval('im(a)')        // Number,  3

// calculations with complex numbers
parser.eval('a + b')        // Complex, 6 + 2i
parser.eval('a * b')        // Complex, 11 + 10i
parser.eval('i * i')        // Number,  -1
parser.eval('sqrt(-4)')     // Complex, 2i
```

Math.js does not automatically convert complex numbers with an imaginary part of zero to numbers. They can be converted to a number using the function `number`.

```
// convert a complex number to a number
const parser = math.parser()
parser.eval('a = 2 + 3i')   // Complex, 2 + 3i
parser.eval('b = a - 3i')   // Complex, 2 + 0i
parser.eval('number(b)')    // Number,  2
parser.eval('number(a)')    // Error: 2 + i is no valid number
```

Units

math.js supports units. Units can be used in the arithmetic operations add, subtract, multiply, divide, and exponentiation. Units can also be converted from one to another. An overview of all available units can be found on the page Units.

Units can be converted using the operator `to` or `in`.

```
// create a unit
math.eval('5.4 kg')                    // Unit, 5.4 kg

// convert a unit
math.eval('2 inch to cm')              // Unit, 5.08 cm
math.eval('20 celsius in fahrenheit')  // Unit, ~68 fahrenheit
math.eval('90 km/h to m/s')            // Unit, 25 m / s

// convert a unit to a number
// A second parameter with the unit for the exported number must be provided
math.eval('number(5 cm, mm)')          // Number, 50

// calculations with units
math.eval('0.5kg + 33g')               // Unit, 0.533 kg
math.eval('3 inch + 2 cm')             // Unit, 3.7874 inch
math.eval('3 inch + 2 cm')             // Unit, 3.7874 inch
math.eval('12 seconds * 2')            // Unit, 24 seconds
math.eval('sin(45 deg)')               // Number, 0.7071067811865475
math.eval('9.81 m/s^2 * 5 s to mi/h')  // Unit, 109.72172512527 mi / h
```

Strings

Strings are enclosed by double quotes " or single quotes '. Strings can be concatenated using the function `concat` (not by adding them using `+` like in JavaScript). Parts of a string can be retrieved or replaced by using indexes. Strings can be converted to a number using function `number`, and numbers can be converted to a string using function `string`.

When setting the value of a character in a string, the character that has been set is returned. Likewise, when a range of characters is set, that range of characters is returned.

```
const parser = math.parser()

// create a string
parser.eval('"hello"')                      // String, "hello"

// string manipulation
parser.eval('a = concat("hello", " world")')  // String, "hello world"
parser.eval('size(a)')                      // Matrix [11]
parser.eval('a[1:5]')                       // String, "hello"
parser.eval('a[1] = "H"')                   // String, "H"
parser.eval('a[7:12] = "there!"')           // String, "there!"
parser.eval('a')                            // String, "Hello there!"

// string conversion
parser.eval('number("300")')                // Number, 300
parser.eval('string(300)')                  // String, "300"
```

Strings can be used in the `eval` function, to parse expressions inside the expression parser:

```
math.eval('eval("2 + 3")')  // 5
```

Matrices

Matrices can be created by entering a series of values between square brackets, elements are separated by a comma `,`. A matrix like `[1, 2, 3]` will create a vector, a 1-dimensional matrix with size `[3]`. To create a multi-dimensional matrix, matrices can be nested into each other. For easier creation of two-dimensional matrices, a semicolon `;` can be used to separate rows in a matrix.

```
// create a matrix
math.eval('[1, 2, 3]')                              // Matrix, size [3]
math.eval('[[1, 2, 3], [4, 5, 6]]')                 // Matrix, size [2, 3]
math.eval('[[[1, 2], [3, 4]], [[5, 6], [7, 8]]]')   // Matrix, size [2, 2, 2]

// create a two dimensional matrix
math.eval('[1, 2, 3; 4, 5, 6]')                     // Matrix, size [2, 3]
```

Another way to create filled matrices is using the functions `zeros`, `ones`, `identity`, and `range`.

```
// initialize a matrix with ones or zeros
math.eval('zeros(3, 2)')      // Matrix, [[0, 0], [0, 0], [0, 0]],  size [3, 2]
math.eval('ones(3)')          // Matrix, [1, 1, 1],                 size [3]
math.eval('5 * ones(2, 2)')   // Matrix, [[5, 5], [5, 5]],          size [2, 2]

// create an identity matrix
math.eval('identity(2)')      // Matrix, [[1, 0], [0, 1]],          size [2, 2]

// create a range
```

```
math.eval('1:4')              // Matrix, [1, 2, 3, 4],             size [4]
math.eval('0:2:10')           // Matrix, [0, 2, 4, 6, 8, 10],      size [6]
```

A subset can be retrieved from a matrix using indexes and a subset of a matrix can be replaced by using indexes. Indexes are enclosed in square brackets, and contain a number or a range for each of the matrix dimensions. A range can have its start and/or end undefined. When the start is undefined, the range will start at 1, when the end is undefined, the range will end at the end of the matrix. There is a context variable `end` available as well to denote the end of the matrix.

IMPORTANT: matrix indexes and ranges work differently from the math.js indexes in JavaScript: They are one-based with an included upper-bound, similar to most math applications.

```
parser = math.parser()

// create matrices
parser.eval('a = [1, 2; 3, 4]')       // Matrix, [[1, 2], [3, 4]]
parser.eval('b = zeros(2, 2)')        // Matrix, [[0, 0], [0, 0]]
parser.eval('c = 5:9')                // Matrix, [5, 6, 7, 8, 9]

// replace a subset in a matrix
parser.eval('b[1, 1:2] = [5, 6]')     // Matrix, [[5, 6], [0, 0]]
parser.eval('b[2, :] = [7, 8]')       // Matrix, [[5, 6], [7, 8]]

// perform a matrix calculation
parser.eval('d = a * b')              // Matrix, [[19, 22], [43, 50]]

// retrieve a subset of a matrix
parser.eval('d[2, 1]')                // 43
parser.eval('d[2, 1:end]')            // Matrix, [[43, 50]]
parser.eval('c[end - 1 : -1 : 2]')    // Matrix, [8, 7, 6]
```

Objects

Objects in math.js work the same as in languages like JavaScript and Python. An object is enclosed by square brackets `{, }`, and contains a set of comma separated key/value pairs. Keys and values are separated by a colon `:`. Keys can be a symbol like `prop` or a string like `"prop"`.

```
math.eval('{a: 2 + 1, b: 4}')         // {a: 3, b: 4}
math.eval('{"a": 2 + 1, "b": 4}')     // {a: 3, b: 4}
```

Objects can contain objects:

```
math.eval('{a: 2, b: {c: 3, d: 4}}')  // {a: 2, b: {c: 3, d: 4}}
```

Object properties can be retrieved or replaced using dot notation or bracket notation. Unlike JavaScript, when setting a property value, the whole object is returned, not the property value

```
let scope = {
  obj: {
    prop: 42
  }
}

// retrieve properties
```

```
math.eval('obj.prop', scope)          // 42
math.eval('obj["prop"]', scope)       // 42

// set properties (returns the whole object, not the property value!)
math.eval('obj.prop = 43', scope)     // {prop: 43}
math.eval('obj["prop"] = 43', scope)  // {prop: 43}
scope.obj                             // {prop: 43}
```

Multi-line expressions

An expression can contain multiple lines, and expressions can be spread over multiple lines. Lines can be separated by a newline character `\n` or by a semicolon `;`. Output of statements followed by a semicolon will be hidden from the output, and empty lines are ignored. The output is returned as a `ResultSet`, with an entry for every visible statement.

```
// a multi-line expression
math.eval('1 * 3 \n 2 * 3 \n 3 * 3')    // ResultSet, [3, 6, 9]

// semicolon statements are hidden from the output
math.eval('a=3; b=4; a + b \n a * b')   // ResultSet, [7, 12]

// single expression spread over multiple lines
math.eval('a = 2 +\n  3')               // 5
math.eval('[\n  1, 2;\n  3, 4\n]')      // Matrix, [[1, 2], [3, 4]]
```

The results can be read from a `ResultSet` via the property `ResultSet.entries` which is an `Array`, or by calling `ResultSet.valueOf()`, which returns the array with results.

Implicit multiplication

Implicit multiplication means the multiplication of two symbols, numbers, or a grouped expression inside parentheses without using the `*` operator. This type of syntax allows a more natural way to enter expressions. For example:

```
math.eval('2 pi')         // 6.283185307179586
math.eval('(1+2)(3+4)')   // 21
```

Parentheses are parsed as a function call when there is a symbol or accessor on the left hand side, like `sqrt(4)` or `obj.method(4)`. In other cases the parentheses are interpreted as an implicit multiplication.

Math.js will always evaluate implicit multiplication before explicit multiplication `*`, so that the expression `x * y z` is parsed as `x * (y * z)`. Math.js also gives implicit multiplication higher precedence than division, *except* when the division matches the pattern `[number] / [number] [symbol]` or `[number] / [number] [left paren]`. In that special case, the division is evaluated first:

```
math.eval('20 kg / 4 kg')   // 5      Evaluated as (20 kg) / (4 kg)
math.eval('20 / 4 kg')      // 5 kg   Evaluated as (20 / 4) kg
```

The behavior of implicit multiplication can be summarized by these operator precedence rules, listed from highest to lowest precedence:

- Function calls: `[symbol] [left paren]`

- Explicit division `/` when the division matches this pattern: `[number] / [number] [symbol]` or `[number] / [number] [left paren]`
- Implicit multiplication
- All other division `/` and multiplication `*`

Implicit multiplication is tricky as there can appear to be ambiguity in how an expression will be evaluated. Experience has shown that the above rules most closely match user intent when entering expressions that could be interpreted different ways. It's also possible that these rules could be tweaked in future major releases. Use implicit multiplication carefully. If you don't like the uncertainty introduced by implicit multiplication, use explicit `*` operators and parentheses to ensure your expression is evaluated the way you intend.

Here are some more examples using implicit multiplication:

Expression	Evaluated as	Result
(1 + 3) pi	(1 + 3) * pi	12.566370614359172
(4 - 1) 2	(4 - 1) * 2	6
3 / 4 mm	(3 / 4) * mm	0.75 mm
2 + 3 i	2 + (3 * i)	2 + 3i
(1 + 2) (4 - 2)	(1 + 2) * (4 - 2)	6
sqrt(4) (1 + 2)	sqrt(4) * (1 + 2)	6
8 pi / 2 pi	(8 * pi) / (2 * pi)	4
pi / 2 pi	pi / (2 * pi)	0.5
1 / 2i	(1 / 2) * i	0.5 i
8.314 J / mol K	8.314 J / (mol * K)	8.314 J / (mol * K)

Comments

Comments can be added to explain or describe calculations in the text. A comment starts with a sharp sign character `#`, and ends at the end of the line. A line can contain a comment only, or can contain an expression followed by a comment.

```
const parser = math.parser()

parser.eval('# define some variables')
parser.eval('width = 3')                              // 3
parser.eval('height = 4')                             // 4
parser.eval('width * height   # calculate the area')  // 12
```