

Expr Language Reference

Expr language defines expressions, which are evaluated in the context of an item in some structure. This article describes the syntax of the language and the rules that govern the evaluation.

- [Conventions](#)
- [Values](#)
 - [Undefined](#)
 - [Text](#)
 - [Numbers](#)
 - [Text to number conversion](#)
 - [Falsy and truthy values](#)
- [Variables and functions](#)
 - [Identifiers](#)
 - [Variables](#)
 - [Function calls](#)
- [Single-argument operators](#)
 - [NOT](#)
 - [+ -](#)
- [Logical and arithmetic operators](#)
 - [Logical operators](#)
 - [Comparison operators](#)
 - [Equality: = \(==\)](#)
 - [Inequality: <> \(!=\)](#)
 - [Ordering](#)
 - [Arithmetic operators](#)
- [Precedence of operators](#)
- [Railroad diagrams](#)
 - [expression](#)
 - [arithmetic-expression](#)
 - [simple-expression](#)

Conventions

- Similarity to Excel formula language was a design goal, so if you're unsure how Expr behaves, think Excel.
- The language is case-insensitive.
- Whitespace is not meaningful. It is only required to separate word operators and identifiers, in all other cases there can be arbitrary number of whitespace symbols.
- Currently language constructs support only English letters and a few punctuation symbols. However, values can contain any Unicode symbols.

Values

All expressions, when evaluated, produce either a value or an error. All values in Expr are either numbers, text, or a special value called *undefined*. The simplest expression thus is the literal representation of some fixed value. The forms of these literal representations are described below.

Undefined

Undefined value is represented by the word `undefined`.

Undefined value is used when the variable value is not specified. For example, variable `Assignee` has value `undefined` if the issue is unassigned.

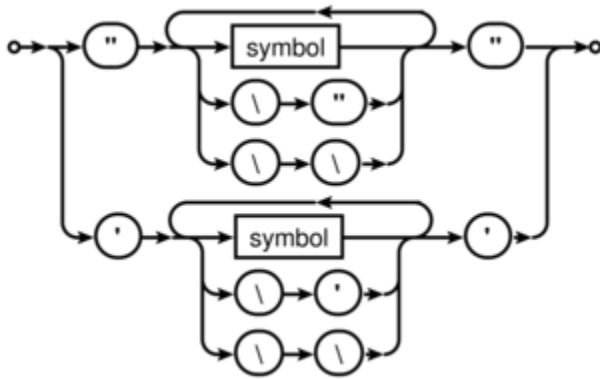
Functions can return this value when the result of function is not specified. For example, function `IF` returns its second argument if the first argument evaluates to a truthy value (see below on that). Otherwise, it returns third argument, but if it wasn't specified, it returns `undefined`.

Text

A text value consists of 0 or more Unicode symbols. Its literal representation consists of the value enclosed in single quotes (') or double quotes ("). Example: `"Major"` represents text value *Major*. Similarly, `'Major'` represents the same text value.

If the text value itself contains quotes, you'll need to insert a backslash (\) before them. Example: `"Charlie \"Bird\" Parker"` represents text value *Charlie "Bird" Parker*. Alternatively, you can use another kind of quotes to enclose the literal representation: `'Charlie "Bird" Parker'`.

If you need to use the backslash at the end of text value, you'll need to insert another backslash before it. Example: `"C:\Users\John\"` represents text value *C:\Users\John*.



Numbers


Aside from representing some quantity, a number value can also represent points in time and duration of time. Then you can use Format settings in the [Formula Column](#) to properly display them as dates or durations.

There are two forms of literal representations of numbers:

- a whole number: 42
- a fractional number: 0.239

Note that only dot (.) can be used as a decimal separator. Comma (,) is used to delimit function arguments. Thus, `MAX(X, 0, 618)` will be understood as the maximum of three quantities: `X`, 0, and 618.

Group separators are not supported, so `100 000` is not a literal representation of number 100000.



Technical note: internally, numbers are represented as decimal floating-point numbers with 16 digits of precision and half-even rounding. Most of the operations are carried out in this form, however, some of the more sophisticated functions, such as `SQRT`, might first convert the numbers into binary floating-point, calculate the result, and then convert it back into decimal floating-point.

Text to number conversion

Some functions expect their arguments to be number values. In case an argument is a text value, we try to interpret it as a number. This can be useful if the value comes from a variable that represents a text custom field, which contains numbers — e.g., imported from some external system.

If conversion is successful, that number is used as the value for that argument. If conversion is not successful, functions can either produce an error, ignore that argument, or substitute some default — it depends on the function; see [Expr Function Reference](#) for details.

The first step is to accommodate for variations in number formatting. Conversion supports these formatting symbols:

- decimal fraction separators:

comma	,
dot	.

- digit group separators:

comma	,
dot	.
apostrophe	'
space	

Conversion expects that the text contains 0 or 1 decimal mark, and 0 or more group separators of the same kind. If text contains any other formatting symbols, conversion fails. Decimal mark must come after all group separators, otherwise conversion fails.

If text contains only one formatting symbol, and it's a dot (.), it is always treated as decimal mark. If text contains only one formatting symbol, and it's a comma (,), then it is treated as decimal mark if comma is decimal separator mark in [JIRA default language](#); otherwise, it is treated as group separator.

After determining decimal mark and group separator symbols, conversion removes all group separator symbols and replaces decimal mark with dot. Note that if text contains several whole numbers separated by spaces, conversion will think that it's one number, for example, `"10 11 12"` will become 101112. Similarly, `"10,11,12"` will become 101112.

The final step of conversion is to recognize the resulting text as either Expr's literal number representation or scientific or engineering notation. Examples:

0.239
-1.32e5
12e-3

Falsy and truthy values

A value is *falsy* if it is:

- undefined,
- number 0,
- empty text value (" " or ' '), or a text value that contains only space characters.

All other values are *truthy*. By convention, when predefined functions or logical operators need to construct a truthy value, they use number 1.

Variables and functions

Other kind of expressions are variables and function calls.

Identifiers

An identifier consists of letters (English only: a-z, A-Z), digits (0-9), dot (.) or underscore (_) characters. The first character must be a letter or an underscore.

Variables

Variables are represented by identifiers. Each variable is resolved to a value once during expression evaluation. If the variable cannot be resolved, its value is `undefined`.

Conceptually, you can think of variable as the cell of some column for the item, in the context of which the expression is evaluated. As such, it might or might not have a value, and that value can be either textual or numeric. Variables are defined in the [Formula Column](#) settings.

Function calls

A function consumes zero or more values, and can produce a value. A function call consists of a function name (an identifier), followed by its arguments enclosed in parentheses. An argument can be any expression. Different arguments are separated by commas (,) or semicolons (;) — for one function call, all separators must be the same.

Function call can evaluate only some or even none of the arguments, depending on the function. This is useful for functions that perform choices, such as `IF`: the argument that wasn't chosen is not evaluated, so the whole expression doesn't produce an error when it produces an error.

Single-argument operators

Expression with single-argument (or *unary*) operator has the following syntax: `<op> <expression>`.

`<expression>` can be any Expr language expression in parentheses. If it is a literal value representation, a variable, or a function call, parentheses are optional.

If `<expression>` evaluation produces error, the operator also produces error.

NOT

Instead of `NOT`, exclamation mark (!) can also be used.

The operator produces 0 if `<expression>` evaluates to a truthy value, and 1 otherwise.

+ -

The operator first attempts to convert the value of `<expression>` to number. If conversion succeeds, + produces this number, and - produces the negated number. If conversion fails, and the value of `<expression>` is falsy, produces `undefined`. Otherwise, produces error.

Logical and arithmetic operators

Two or more expressions can be combined using operators: `<expression1> <operator> <expression2>`. If any subexpression produces error, the operator produces the same error.

Logical operators

OR (||, |)

AND (&&, &)

OR examines each expression from left to right, and produces the value of the first expression that evaluates to a truthy value. If no expression evaluates to a truthy value, returns `undefined`. All expressions that come after the first that evaluates to a truthy value are not evaluated. This prevents from unnecessary computations, and protects from producing error if any of the subsequent expression produces an error.

AND works in the same way. The only difference is that it looking for the first *false* value.

Examples (assuming the default variable assignment):

- `assignee || "UNASSIGNED"` will produce either issue's assignee user key or text value "UNDEFINED" if the issue is unassigned.
- `!assignee && status = "OPEN"` will produce 1 if the issue is unassigned and in status OPEN, and 0 otherwise.

Comparison operators

All of these operators produce 0 or 1. These operators can work only on two arguments. They start with evaluating both expressions. All comparison operators have the same precedence.

Equality: = (==).

If both values are numbers, returns 1 if they are equal.

If both values are text, returns 1 if they are equal, ignoring differences in letter forms and leading and trailing whitespace (thus `" cote "` = `"côte"`).

If both values are undefined, returns 1.

In all other cases returns 0.



If one value is a number and the other value can be converted to a number, both values are treated as numbers. However, if both values are text, they will be treated as text, even if both can be converted to a number. You can use [NUMBER](#) function to force a value to be numeric.

- `3.4 = 3.40` 1
- `3.4 = "3.40"` 1
- `"3.4" = "3.40"` 0
- `NUMBER("3.4") = "3.40"` 1

Inequality: <> (!=)

Works in the same way as equality operator, but returns 0 where it returns 1 and vice versa.

Ordering

< (less than)

> (greater than)

<= (less than or equal)

>= (greater than or equal)

All operators work on numbers, producing the result of their comparison.

If either of the values is text, attempts to convert it to number. If conversion fails, operators behave as if the corresponding value was undefined.

If any value is undefined, strict operators (<, >) produce 0. Non-strict (<=, >=) produce 0, unless *both* values are undefined (because they are equal).

Arithmetic operators

Arithmetic operators are: addition (+), subtraction (-), multiplication (*) and division (/).

These operators convert their arguments to numbers. A non-empty non-number argument would produce an error. Falsy non-number values are treated as zero.

Examples:

- `" " + 1` 1
- `"foo" + 1` error
- `" " * 1` 0
- `"foo" * 1` error
- `" " - 1` -1

- 1/0 error

Precedence of operators

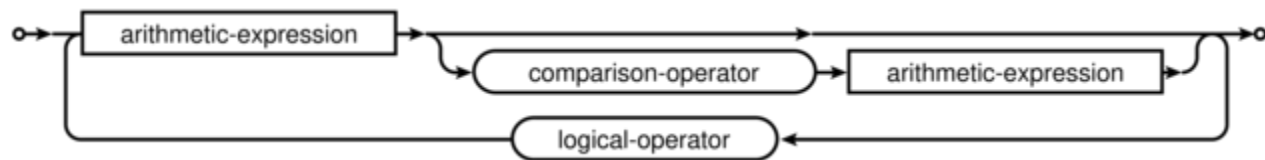
Precedence defines which operators evaluate first: if operator A has lesser precedence than B, then in expression $\langle \text{expression1} \rangle A \langle \text{expression2} \rangle B \langle \text{expression3} \rangle$ first B is evaluated, then A.

Single-argument operators are always evaluated first. Other operators in Expr language have the following precedence:

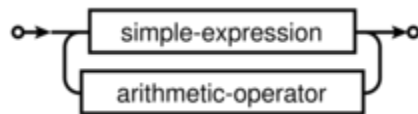
1 (lowest)	OR
2	AND
3	= <> < > <= >=
4	+ -
5 (highest)	* /

Railroad diagrams

expression



arithmetic-expression



simple-expression

