

Expr Language Reference

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

- [Conventions](#)
 - [Comments](#)
- [Values](#)
 - [Undefined](#)
 - [Text](#)
 - [Numbers](#)
 - [Text to Number Conversion](#)
 - [Falsy and Truthy Values](#)
- [Variables and functions](#)
 - [Identifiers](#)
 - [Variables](#)
 - [Local Variables](#)
 - [Function Calls](#)
 - [Aggregate 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](#)
 - [logical-expression](#)
 - [arithmetic-expression](#)
 - [function-call](#)
 - [aggregation-expression](#)

Conventions

- Similarity to Excel formula language was a design goal, so if you are 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 an arbitrary number of whitespace symbols.
- Currently, language constructs support only English letters and a few punctuation symbols. However, values can contain any Unicode symbols.

Comments

At any place where a formula allows whitespace, you can use comments. Comments can span multiples lines or just one.

- Multi-line comments start with `"/*"` and end with `"*/"` and can span multiple lines. Multi-lined comments cannot be nested.
- Single-line comments start with `"//"` and continue through the end of the line.

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*.

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.

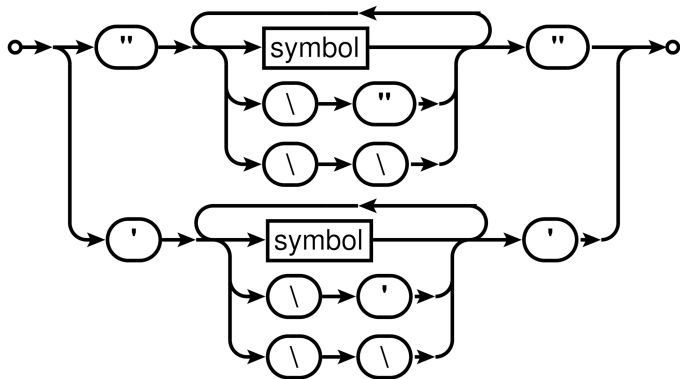
Functions can return this value when the result of the function is not specified. For example, the function `IF(N = 0; "No apples"; N = 1; "One apple")` only has a specified value when N is equal to 0 or 1. If N is equal to anything else, it returns `undefined`.

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 the 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 will 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 a point in time or a duration of time. In this case, you can use Format settings in the [Formula Columns](#) to properly display the results 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 the text contains any other formatting symbols, conversion fails. Decimal mark must come after all group separators, otherwise conversion fails.

If the text contains only one formatting symbol, and it's a dot (.), it is always treated as a decimal mark. If the text contains only one formatting symbol, and it's a comma (,), then it is treated as a decimal mark if a comma is used as a decimal separator mark in the [Jira default language](#); otherwise, it is treated as a group separator. For instance, if the default Jira language is English, "101,112" will become 101112, whereas if it is German locale, it will be 101.112. And regardless of language, "1 100,23" will become 1100.23: space is interpreted as a group separator, and comma can only be the decimal fraction separator here.

If the group separator is a dot (.), then all groups except the first one must have 3 digits; otherwise, conversion fails.

After determining decimal mark and group separator symbols, conversion removes all group separator symbols and replaces the decimal mark with a dot. Note that if text contains several whole numbers separated by spaces, conversion will think it is 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 kinds of expressions are variables and function calls.

Identifiers

An identifier consists of letters (Latin alphabet 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. Variables are defined in the [Formula Columns](#) settings and are mapped to a Jira field or other attribute of an item.

Conceptually, you can think of a 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. Each variable is resolved to a value once during the expression evaluation. If the variable cannot be resolved, its value is *undefined*.

Local Variables

Local variables are similar to Variables, but they are not mapped to the item's attribute or Jira field, but rather defined and calculated right in the expression.

The declaration syntax is the following:

```
WITH <local_variable_name> = <expression> : <expression_with_local_variable>
```

Note the colon (":") that separates the expression assigned to the variable and the expression where the variable is used.

A few facts about local variables:

- *<expression_with_local_variable>* may start with another local variable definition, so you can introduce many local variables. When defining a second variable, you can use the first variable already defined, and so on.
- Local variables can "shadow" previously defined local and free (mapped) variables with the same name. If you write "with priority = 10: <expression>", then when calculating "<expression>", the value of `priority` will be 10, even if there was a variable attached to the issue's priority in the enclosing scope.
- The `with...` construct is itself an expression, so you can use it, enclosed in parentheses, anywhere an expression can be used. The name defined in this expression is not visible outside the `with...` expression.

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.

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

Aggregate Function Calls

An aggregate function call takes an expression and calculates it for all sub-items (or for another sub-set of the structure, as defined in the function's documentation).

An aggregate function may have one or more modifiers that govern aspects of function execution. Each modifier starts with hash sign ("#"), then comes the name (an identifier), an optional equal sign ("=") and a value, which can be a string or numeric constant. If a value is omitted, it is assumed to be 1 (a representation of *true* in Expr).

An aggregate function must be followed by the expression in curly braces ("{}"), which provides the values being aggregated.

You can use whitespace between any elements of the aggregate function calls.

Examples:

- `SUM{x}`
- `SUM#all{x}`
- `SUM#all#leaves{x}`
- `JOIN#separator=", " {key}`
- `JOIN #separator=", " #fromDepth=0 #toDepth=-1 { Key }`

Not all modifiers will work with every aggregate function. Using an incompatible modifier will result in an error. To learn more about available modifiers and their restrictions, see [Aggregation Modifiers](#).

Single-argument operators

Expressions with a single-argument (or *unary*) operator have 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 an error, the operator also produces an error.

NOT

Instead of NOT, an 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 a number. If conversion succeeds, + produces this number, and - produces the negated number. If conversion fails, and the value of `<expression>` is falsy, it produces *undefined*. Otherwise, it produces an error.

Logical and arithmetic operators

Two or more expressions can be combined using operators: `<expression1> <operator> <expression2>`. If any subexpression produces an 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, it returns *undefined*. Once a truthy expression is found, no other expressions are evaluated. This prevents unnecessary computations and protects against producing an error if any of the subsequent expressions produce an error.

AND works in much the same way, except that it is looking for the first *falsy* value.

Examples (assuming the default variable assignment):

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

Comparison operators

All comparison operators:

- Produce 0 or 1

- Can work only on two arguments
- Start with evaluating both expressions
- 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 the [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 the equality operator, but returns 0 when both values are equal or undefined, 1 when they are not.

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, the operator attempts to convert it to number. If the conversion fails, the operator behaves 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 <expression1> A <expression2> B <expression3> first B is evaluated, then A.

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

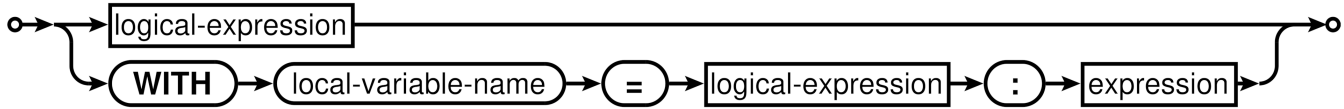
5 (highest)	* /
4	+ -
3	= <> < > <= >=

2	AND
1 (lowest)	OR

Railroad diagrams

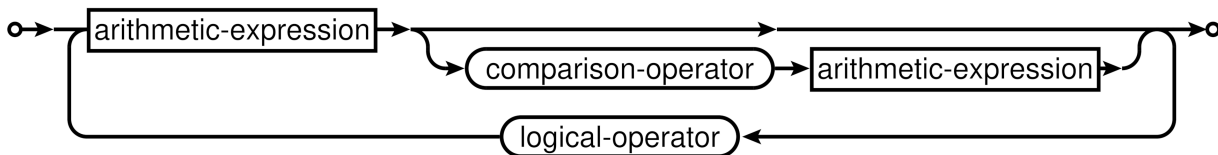
These diagrams display the complete syntax of Expr language.

expression



local-variable-name is an [identifier](#).

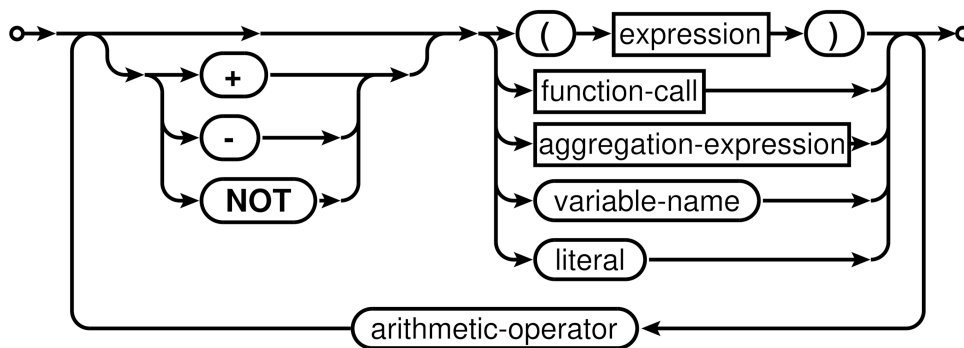
logical-expression



comparison-operator is one of these: = <> < > <= >=.

logical-operator is one of these: AND OR.

arithmetic-expression

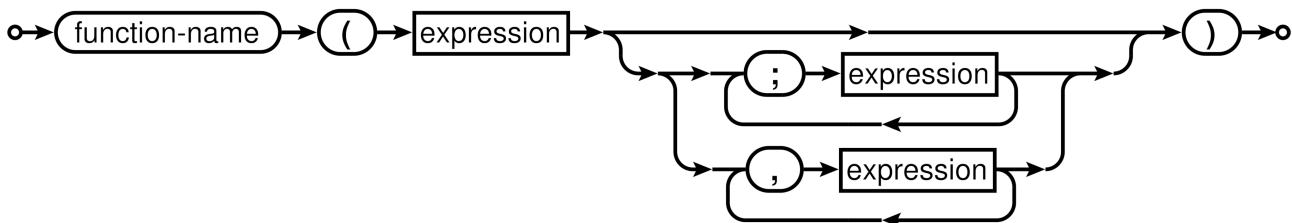


variable-name is an [identifier](#).

literal is either a [number literal](#), a [text](#) or [UNDEFINED](#).

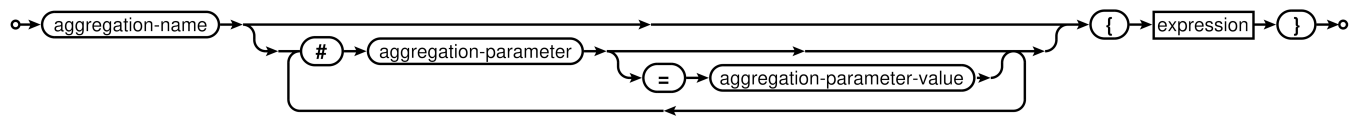
arithmetic-operator is one of these: + - * /.

function-call



function-name is an [identifier](#).

aggregation-expression



aggregation-name and aggregation-parameter are **identifiers**.

aggregation-parameter-value is either a **text** or a **number literal** with optional sign (either + or -).