# Expr Language

**Expr Language** (pronounced like "expert" without "t") is a language that lets you specify an "expression", or a formula, which will be calculated for an issue or another item. When you use is in a Formula Column, the expression is calculated for each visible row in the displayed structure or query result.

You can see examples of formulas by adding predefined columns in Structure (from the **Calculated** section), and then opening column options panel. The language itself and its grammar is quite obvious and is similar to arithmetic expressions with a number of functions.

If you're familiar with writing formulas in **Microsoft Excel**, you'll recognize a few things in Expr. In particular, if you know some functions that Excel provides, they have a good chance to be supported by Structure as well.

## Language Components

An expression may contain one or more of the following:

- Variables, which are bound to specific values when calculating expression for a specific item.
- Functions, which may take some arguments, and which produce the result at the moment of calculation.
- Numbers and text strings.
- Arithmetic, logical operations and parentheses.

There are also more advanced constructs:

- Aggregate Functions, which calculate some aggregate (like sum or average) of an expression's values calculated for multiple items in the structure.
- Local Variables, which let you introduce a value and reuse it multiple times in the formula.
- Comments, which allow you document larger formulas.

## Basic Constructs

### Variables

Variables are user-defined names, containing letters (English only), numbers, dot (".") or underscore ("_") characters. The first character should be a letter or an underscore.

Examples:

- `Priority`
- `remaining_estimate`
- `abc11`
- `sprint.name`

When writing an expression, you'd usually associate a name with some value that an issue or another item has. After the expression is written, Formula Column lets you associate the variables used with specific attributes.

A few things to note about variables:

- You can use any names, from a simplistic "x" to a "VeryComplicatedCustomFieldName".
- But, if Formula Column recognizes the variable name to be similar to a field name, it may automatically assign an attribute to the variable. For example, "Priority" variable will be automatically mapped to the similarly named field.
- But, it is possible (although very unreasonable!) to edit the association and assign a variable with a well-known name to something else. ⚠️Please keep that in mind if you need to troubleshoot a formula and double-check the variables.

Variable names are case-insensitive. `Priority`, `priority` and `pRiOrItY` will refer to the same variable.

### Functions

A function calculates some value based on its arguments and, sometimes, some external aspect. A function is written as the function name, followed by parentheses, which might contain arguments.

Examples:

- `SUM(-original_estimate; remaining_estimate; time_spent)`
- `CASE(priority, 'High*', 5, 1)`
- `TODAY()`

There are a number of standard functions available with Structure 4.0 – see Expr Function Reference for details.

A function may take zero, one or more arguments. Some functions take variable number of arguments. Each argument can be another Expr expression and include calls to other functions.

Function arguments may be separated by comma (,) or semicolon (;). But in every function call you need to use either all commas or all semicolons.

Function names are case-insensitive, like the variables. You can write `TODAY()` or `Today()`.

## Numbers and Text Strings

### Numbers

You can use numbers in your formula. The numbers are always written as a sequence of digits with optionally a dot (".") and a fractional part. Locale-specific, percentage, currency or scientific formats are not supported.

| Recognized as a number | Not recognized as a number |
|---|---|
| `0` | `0,0` |
| `1234567890123456` | `1 100 025` |
| `11.25` | `1.234e+04` |
| `.111` | `($100)` |

You can write a number that is written with a locale-specific decimal and thousands separator as a text value and it will be automatically converted to a number if needed. For example:

- `"1 122,25" * 2  2244.5`

### Text Strings

Text strings are a sequence of characters enclosed either in single (') or double quotes ("). Examples:

- `'a text in single quotes may contain " (a double quote)'`
- `"a text in double quotes may contain ' (a single quote)"`
- `""`

Everything within a text string is retained verbatim to participate in the expression evaluation, except for the following:

- A sequence of two backslashes (`\\`) is converted to a single backslash (`\`).
- A sequence of a backslash and a single quote (`\'`) is converted to a single quote character (`'`) in the text values enclosed in single quotes.
- A sequence of a backslash and a double quote (`\"`) is converted to a double quote character (`"`) in the text values enclosed in double quotes.

## Operations

Expr provides basic arithmetic operations, comparisons and logical operations.

The operations follow the general precedence rules for arithmetic, so `A + B * C` is calculated correctly. Comparison operations are done after the arithmetic operations and logical operations are done after comparisons. For detailed specification, see Expr Language Reference.

| Operations | Comments |
|---|---|
| `+ - * /` | Convert the value into a number. |
| `= !=` | Equality and non-equality: if either part of the comparison is a number, the other part is also converted into a number. If both values are strings, then string comparison is used. String comparison ignores leading and trailing whitespace and is case-insensitive (according to JIRA's system locale). |
| `< <= > >=` | Numerical comparisons – both values are converted to numbers. |
| `AND`, `OR`, `NOT` | Logical operations. |
| `( )` | Parentheses can be used to group the results of operations prior to passing to other operations. |

## Advanced Constructs

## Aggregate Functions

An aggregate function calculates some aggregate value (like sum or minimum) based on values in a number of rows, typically, for all sub-issues.

Examples:

- `SUM{ remaining_estimate + time_spent }` – calculates the total effort (estimated and actual) for the issue and all sub-issues.
- `MAX{ resolved_date - created_date }` – calculates the maximum time it took to resolve an issue, among the issue and its sub-issues.

The list of available Aggregate Functions is available in Aggregate Function Reference.

Aggregate function contains exactly one expression that is being aggregated, written in curly braces (`{}`) after the function name.

It can also contain **modifiers**, which influence how the aggregation works:

- `SUM#all{ business_value }` – this will force the function to include values from all duplicate items in the total. (By default, duplicates are ignored.)

✓ Note that there is `SUM()` function and `SUM{}` aggregate function. You can always tell aggregate functions from the usual functions by the use of curly braces (like `SUM{x}`).

## Local Variables

Local variables help when some expression needs to be used in the same formula several times. For example:

-
  ```
  IF(time_spent + remaining_estimate > 0;
     time_spent / (time_spent + remaining_estimate))
  ```

You can see that in this formula we are using "`time_spent + remaining_estimate`" two times – one time when we check that it's not zero (so we don't divide by zero) and then when we divide by it.

It's possible to rewrite this formula using **`WITH`** construct:

-
  ```
  WITH total_time = time_spent + remaining_estimate :
  IF(total_time > 0; time_spent / total_time)
  ```

You can define multiple local variables in succession, following one `WITH` declaration with another. You can use previously defined local variables when defining local variables that follow. Example:

-
  ```
  WITH total_time = time_spent + remaining_estimate :
  WITH progress = IF(total_time > 0; time_spent / total_time) :
  IF(progress > 0.5; "Great Progress!"; progress > 0.2; "Good Progress"; "Needs Progress")
  ```

✓ Note the position of colon ("`:`") – it must be present where local variable definition ends.

## Comments

Comments are helpful when you have a large formula or when a reader might need explanations of what is calculated. It's a good idea to add comments where the formula is not trivial.

Example:

```
/* This formula calculates the verbal assessment of issue's progress.
   And this explanation is a comment that spans multiple lines. */

WITH total_time = time_spent + remaining_estimate :

// Progress is calculated based on time tracking. (This is a one-line comment.)
WITH progress = IF(total_time > 0; time_spent / total_time) :

IF(progress > 0.5; "Great Progress!"; progress > 0.2; "Good Progress"; "Needs Progress")
```

## See Also

- Expr Function Reference

- Expr Language Reference