Manual 07 · expressions, scripts, functions

FPEL Language Reference

FPEL is the expression language of Stilton. The same expressions compute fields in Transform and TMap, decide routes in Filter, check data in Data Quality, and make up the statements of the Script node. This reference covers the language, its 107 built-in functions, user functions, decode tables and the ready-made templates.

CONTENTS

01 **One language, two levels**
- 1.1 Where you write FPEL

02 **Expressions**
- 2.1 Literals
- 2.2 Fields
- 2.3 Lane variables
- 2.4 Counters
- 2.5 Operators
- 2.6 Constructs
- 2.7 Functions on list elements
- 2.8 Decode tables

03 **User functions**
- 3.1 "Explain this value"

04 **The Script node**
- 4.1 Statements
- 4.2 Examples
- 4.3 Rules that matter
- 4.4 The Script editor

05 **Types**

06 **Field lineage**

07 **Filters pushed into database queries**

08 **Built-in functions**
- 8.1 Text (30)
- 8.2 Numbers (14)
- 8.3 Dates and times (32)
- 8.4 Conversion, encoding, hashing (12)
- 8.5 Logic and null (3)
- 8.6 Lists (10)
- 8.7 Structures (JSON objects and arrays) (6)
- 8.8 Lane variables (4)
- 8.9 Date formats

09 **Templates**
- 9.1 Script templates
- 9.2 Base
- 9.3 Multiple rows
- 9.4 Strings
- 9.5 Date
- 9.6 Numbers
- 9.7 Checks
- 9.8 Lane variables
- 9.9 Inline templates by field type
- 9.10 Text fields (38)
- 9.11 Number fields (24)
- 9.12 Integer fields (11)
- 9.13 Decimal fields (11)
- 9.14 Boolean fields (8)
- 9.15 Date fields (22)
- 9.16 Object fields (7)
- 9.17 any (9)

Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 One language, two levels

Statements — Script node only

assign · let · if / else · for · repeat · emit · skip · reject · log · error · expect

Expressions — everywhere a value or a condition is written

```
price * 1.22 upper(name) age ⩾ 18 and status == "active"
```

Transform · TMap · Filter · Data Quality · Script · user functions

- **Expressions** produce one value: `price * 1.22`, `upper(name)`, `case when age ⩾ 18 then "adult" else "minor" end`. Every node that computes on a field evaluates expressions.
- **Statements** exist only in the **Script** node. They add what an expression cannot say: assign several fields, branch, loop, emit more rows, decide that a row does not go out. Conditions and values inside statements are, again, expressions.

Note. An invalid expression is an error, never a silent degradation. A misspelt function name or a wrong number of arguments is reported in the studio, with its position, before you can run — it never reaches the data.

The studio parses and checks FPEL; the engine receives the compiled form and never sees the text. What you see validated is exactly what runs.

Where you write FPEL

Node	What you write	Example
Transform	one expression per output field	`price_vat = round(price * 1.22, 2)`
TMap	one expression per output column; filters per output	`concat_ws(" ", Customers.first_name, Customers.last_name)`
Filter	one condition per route	`amount > 0 and customer_id is not null`
Data Quality	rules on fields	`email like "%@%.%"`
Script	a body of statements	see chapter 4

02 Expressions

Literals

Kind	Examples
Numbers	`42`, `3.14`, `.5`, `1_000_000` (underscores for readability)
Text	`"Rome"` or `'Rome'`; escapes `\"`, `\'`, `\\`, `\n`, `\t`
Booleans	`true`, `false`
Null	`null`
Text with placeholders	`f"Order {id} for {customer}"` — each `{expression}` is replaced by its value; null becomes empty text; `{{` and `}}` are literal braces
Lists	`[1, 2, 3]`, `["IT", "FR"]`, `[]`

FPEL — placeholders text

```
f"Total: {round(qty * price, 2)} EUR"            // "Total: 1234.5 EUR"
f"{last_name}, {first_name} ({country ?? "n/a"})"
```

Fields

A bare name is a **field of the row**: `price`, `name`, `città` (identifiers are Unicode). Names with spaces or punctuation go between backticks, as in SQL: `` `order date` ``, `` `cost/unit` ``.

In nodes with **several inputs** (the TMap) a field is qualified with the input's label: `Customers.name`, or `"Order lines".amount` when the label has spaces.

Lane variables

`var("name")` reads a variable. At run time a lane sees its own variables and the shared variables of the pool, with the active profile's value; a lane variable with the same name wins. Expressions can read variables anywhere; only the Script can write them (chapter 4).

FPEL text

```
amount ⩾ to_float(var("THRESHOLD"))
f"{var("OUT_DIR")}/report_{date_format(today(), "yyyy-MM-dd")}.csv"
```

Counters

`next("name")` gives the next number of a counter: 1, 2, 3… It is the simple way to number rows, build a programmatic key or a composite code, without arithmetic on variables and without worrying about types: the result is **always a whole number**.

Function	Gives
`next("name")`	advances the counter and returns the new value
`current("name")`	the last value given by `next`, without advancing; null before the first `next`
`reset("name")` · `reset("name", 100)`	restarts the counter: the next `next` returns the start value, or 100

FPEL — a composite key text

```
concat(main.year, "-", main.branch, "-", lpad(next("prog"), 6, "0"))
// 2026-MI-000001, 2026-MI-000002, …
```

Each node has its own sequence. Two nodes that call `next("prog")` count independently, so the same input always gets the same numbers. When other nodes need the number, it travels in the row as a field — through a bridge too. Validation warns when two nodes of the same lane advance the same counter: their numbers would repeat. To share one sequence, generate the number once (after a Union, or before splitting the flow) and pass it on; to keep two, give them different names or a prefix (`concat("A-", next("prog"))`).

How a counter counts is declared in the lane. Add a variable of type **counter** among the lane variables (select the lane, tab *Lane*):

Setting	Meaning	Default
value	start: the first number `next` returns	empty = the shared (pool) variable with the same name, if any, otherwise 1
step	added at every `next`; also negative, never 0	1
reset	*restart every run*, or *restart every session* of a continuous flow (Dir Watcher, FTP/SFTP)	every run

A counter that is not declared works anyway: it starts from 1 and counts by 1 — or starts from the value of a numeric variable with the same name. Because the start may come from a **shared** variable, it can be made configurable on the runner's machine: 1 in test, 500 000 in production, with the same artifact.

Tip. The form `lane.prog++` used in TMap transforms in earlier versions is the same as `next("prog")`.

Counters in the TMap

Where you write `next` decides when it advances. With a lookup that has several matches, one main row produces several *combinations*:

Orders o1 (lines A, X) and o2 (lines B, C, D); the output filter drops line X

combination	in a transform	transform, per main row	in an output field
o1 · A	1	1	1
o1 · X (filtered)	2	1	—
o2 · B	3	2	2
o2 · C	4	2	3
o2 · D	5	2	4
	same value on every output	one number per order	no gaps, per output

The same next() in three places: once per combination, once per main row, once per emitted row.

- **In a transform** (default: *per combination*): computed once for each combination, before the output filters, and shared by all outputs — the same number reaches "headers" and "lines" of the same combination, which is what links them. Filtered combinations, and rejected rows when the rejected output is used, consume a number too.
- **In a transform set to *per main row*** (the button next to the mode in the transform header): computed at the first combination of a main row and reused for the others — one number per order, whatever the number of lines.
- **In an output field**: computed only for the rows that output actually emits — a progressive without gaps, separate for each output.

Operators

From the lowest to the highest precedence:

Level	Operators	Notes
1	`||` `or` `??`	logical OR; `a ?? b` is `a`, or `b` when `a` is null
2	`&&` `and`	logical AND
3	`==` `≠` `in` `between` `like`	equality and membership; `not in`, `not between`, `not like` too
4	`<` `⩽` `>` `⩾`	comparison
5	`+` `-`	`+` **concatenates** when one side is text
6	`*` `/` `%`	multiplication, division, remainder

Unary `!` (or `not`) negates, unary `-` changes sign. Parentheses group.

Comparisons behave like SQL. A missing value — **null** — is neither equal nor different from anything: comparing it gives *unknown*, which FPEL writes as null. A condition (Filter, `if`, `expect`, TMap filters and joins, `case when`) is met **only when it is true**, so *unknown* counts as not met. To ask about a missing value, ask explicitly: `is null`, `is not null`, `is empty`, `??`.

Expression	Result	Why
`amount < 100` with `amount` null	null (not met)	a comparison with null is unknown
`country ≠ "IT"` with `country` null	null (not met)	the same for `≠`: the row does not pass
`country ≠ "IT" or country is null`	true for null countries	say it explicitly when nulls must pass
`amount > 0 and country == "IT"` with `country` null	null (not met) — but **false** if `amount` is 0	`and`: false wins; otherwise unknown

Expression	Result	Why
`amount > 0 or country == "IT"` with `country` null	**true** if `amount` is 5	`or`: true wins; otherwise unknown
`not (amount > 100)` with `amount` null	null (not met)	the opposite of unknown is unknown
`country == null` / `country != null`	true / false for a null country	null written **literally** means `is null` / `is not null`
`items is empty`	true for null, `""` or only spaces (or an empty list)	the everyday "is there anything?" — never null
`(qty ?? 0) > 5`	false for a null qty	`??` gives a value to use when it is missing
`"10" > 9`	true	a number on one side makes the other compare as a number (typical of CSV text)
`"abc" > 9`	false	text that is not a number cannot be compared with a number
`name == "mario"` with `"Mario"`	false	text comparison is case-sensitive: use `lower(name) == "mario"`

Constructs

Construct	Example	Notes
Case	`case when qty > 100 then "large" when qty > 10 then "medium" else "small" end`	at least one `when`
Ternary	`vip ? 0.10 : 0`	same as `iif(vip, 0.10, 0)`
Cast	`cast(code as integer)`	`integer` / `int`, `float` / `double`, `decimal`, `string` / `text`, `boolean` / `bool`, `date`, `datetime`
Null test	`email is null`, `email is not null`	`email == null` is the same as `email is null`
Empty test	`email is empty`, `email is not empty`	null, `""` or only spaces; never null itself
Membership	`status in ("A", "B", "C")`, `"urgent" in tags`	same comparison as `==`; a null `status` is unknown. A `null` inside the list is an error: write `status is null or status in (…)`
Range	`age between 18 and 65`	bounds included
Pattern	`name like "Ma%"`	`%` any sequence, `_` one character, `\%` `\_` literal; case-sensitive
Fallback	`to_int(code) ?? 0`, `a ?? b ?? "none"`	a calculation that fails gives null, never an error

Tip. Because a failed conversion gives null, `to_int(code) ?? 0` reads as *"the code as a number, or 0 if it is not a number"*. `try(to_int(code), 0)` says the same.

Functions on list elements

`filter`, `map`, `any`, `all`, `sum` and `count` accept a **function** as second argument, written `name -> expression`:

FPEL — lines is a JSON array of objects text

```
filter(lines, l → l.qty > 0)            // the lines with qty > 0
map(lines, l → upper(l.sku))            // the SKUs in upper case
sum(lines, l → l.qty * l.price)         // the order total
any(tags, t → t like "urg%")            // at least one tag starting with "urg"
all(scores, s → s ⩾ 6)                  // all passing (true for an empty list)
count(lines, l → l.qty == 0)            // how many lines are at zero
```

- `l.qty` reads the field `qty` of the element; `l.a.b` goes deeper.
- Inside the function, other names are still fields of the row: `any(lines, l → l.qty > min_qty)`.
- Functions nest: `any(a, x → any(b, y → y == x))`.
- A value that is not a list gives null. Without a function, `sum(list)` adds the numbers and `count(list)` counts the non-null elements.

Decode tables

Code-to-description lists of the project, managed with **Tables** in the toolbar. Paste two columns from a spreadsheet, or type them.

countries

`IT`	Italy
`FR`	France
`DE`	Germany

FPEL text

```
decode("countries", country_code)               // "IT" → "Italy", unknown code → null
decode("countries", country_code, "unknown")    // with a fallback
to_float(decode("vat_rates", category))         // values are text: convert when you need a number
```

The table name is written literally and is not case-sensitive; an unknown table is an error in the panel. Keys compare as text (`1` and `"1"` are the same key). Tables are compiled into the expressions: the artifact does not depend on external files, and lookups take constant time.

03 User functions

When the same calculation appears in many places, define it once. **Functions** in the toolbar opens the editor of the project's functions.

Functions text

```
function vat(net, rate) { return net * rate / 100 }
function gross(net, rate) { return net + vat(net, rate) }
function full_name(first, last) { return trim(concat_ws(" ", first, last)) }

example vat(100, 22) == 22
example gross(100, 22) == 122
example full_name(" Ada ", "Lovelace") == "Ada Lovelace"
```

- The body is **one expression** that uses only its parameters; no recursion; a user function cannot hide a built-in one.
- User functions work in Transform, TMap and Script, appear in the function picker and in autocompletion, and are saved in the project.
- They are **expanded at compilation**: the engine runs the expanded expression, so a user function costs nothing at run time.

- **Examples are tests.** Each `example` line must be true. The studio checks them as you type (✓ / × under the editor, with the value obtained), and again at every **Run** and **Compile**: a false example stops both, with the reason in the log.
- **Libraries**: export the functions to a `.ffpel` file and import them into other projects. Examples travel with them.

"Explain this value"

In the Transform preview, click a computed (underlined) value: the engine shows the calculation step by step on that sample row, with the value of every part. Branches that did not need to be computed — a `case` after the chosen branch, a `??` after the first non-null value — are marked *not evaluated*.

04 The Script node

A Script runs **once per input row**. Its output schema is *the input schema plus the fields the body assigns*.

Rows source	Behaviour
Flow (default)	there is an input port; the processed row goes out by itself at the end of the body
Generate	no input port; the node produces rows from nothing, and a row goes out **only** when the body runs `emit`

Statements

Statement	Form	Effect
Assignment	`field = <expr>`	creates or overwrites an output field
Local value	`let name = <expr>`	an intermediate value, visible in the current block
Write a lane variable	`var("name") = <expr>`	private to this node, keeps its value from row to row
Counter	`reset("name" [, v])` · `next("name")`	restarts a counter, or advances it without using the value (chapter 2, *Counters*)
Condition	`if <cond> { … } else if <cond> { … } else { … }`	branching
Repeat	`repeat <n> [as <name>] { … }`	n iterations, with an optional counter
Loop	`for <name> in <list> { … }`	iterates over a list
Emit	`emit`	sends **a copy** of the row as it is now — and continues
Skip	`skip`	the row does not leave the main port
Reject	`reject [<reason>]`	sends the row to the reject port, with `_reject_reason`
Log	`log <expr>`	a log line (info)
Error	`error <expr>`	fails the node with a message
Rule	`expect <cond> else reject | warn | error [<message>]`	a named data rule whose violations are counted

Examples

Script — enrich and route

```text
let discount = vip ? 0.10 : 0
final_price = round(price * (1 - discount), 2)

if final_price > 1000 {
  priority = "high"
  log f"large order {order_id}: {final_price}"
} else {
  priority = "normal"
}

if customer_id is null {
  reject "missing customer"
}
```

Script — one row becomes N (fan-out)

```text
// an order row with a JSON array "lines" becomes one row per line
for l in lines {
  sku    = l.sku
  qty    = l.qty
  amount = round(l.qty * l.price, 2)
  emit
}
skip      // the original order row does not go out; the emitted ones do
```

Script — data rules counted by the monitor

```text
expect quantity > 0          else reject "quantity must be positive"
expect email like "%@%"      else warn   "suspicious e-mail"
expect order_id is not null  else error
```

Script — running total across rows

```text
var("running") = (var("running") ?? 0) + amount
running_total = var("running")
```

Script — line numbers that restart for every document

```text
if document ≠ var("last_document") {
  reset("line")
}
line_no = next("line")          // 1, 2, 3… then 1, 2… for the next document
var("last_document") = document
```

Script — Generate mode: a calendar of 31 days

```text
repeat 31 as i {
  day = add_days(start_of_month(today()), i)
  weekend = is_weekend(day)
  emit
}
```

Rules that matter

- **`emit` is fan-out, not return.** It sends a copy and continues; later statements may change the row and emit it again. Emitted rows go out even if the body ends with `skip` or `reject`.
- **`reject`** needs the node's reject port enabled. **`skip`** simply drops the row. **`error`** fails the node (the error goes to the lane's Error Handler).
- **`let` has block scope.** A `let` inside an `if` lives in that `if`. `let x = x + 1` reads the *field* `x`, because the value is computed before the name exists.
- **`expect`**: a row violates the rule when the condition is not true, including when it is null. `reject` sends the row to the reject port; `warn` lets it through and logs the first violation; `error` fails the node at the first violation. At the end of the node — even if it failed — the engine reports each rule with its violation count: the studio shows `✓ 3 rules` or `⚑ 1/3 rules` on the node, and the monitor in the run summary.
- **Comments** start with `//`. Braces end a logical line: `if z { b = 1 }` on one line is the same as on four.
- **Reserved words**: `let`, `if`, `else`, `skip`, `reject`, `log`, `error`, `emit`, `repeat`, `for`, `in`, `as`.
- **Loop limit**: at most 1 000 000 iterations per row (sum of `repeat` and `for`). There is no `while`.

The Script editor

A full code editor with FPEL highlighting, completion of fields, lane variables and functions, buttons that insert statements and field names, live validation with line numbers, and the ⚡ **templates** menu (chapter 8).

05 Types

Every expression has an inferred type: `string`, `integer`, `decimal`, `boolean`, `date`, `datetime`, `object`, or `any` when it cannot be known. Types decide the output schema — a new Script field takes the type of the expression that produces it.

- Functions that return *depends on arguments* (`min`, `max`, `clamp`, `coalesce`, `nullif`, `iif`) take the type of their arguments: `iif(c, 1, 2)` is an integer, `iif(c, "a", "b")` is text.
- `+` concatenates when one side is text, so `"Order " + 12` is `"Order 12"`. When in doubt, make it explicit with `to_string(…)`.

06 Field lineage

Lineage in the toolbar answers two questions about any field:

- **Where it comes from** — the path back to the sources: each step names the node, the field, *how* it is produced (*source*, *passed*, *computed*, *aggregated*…) and the expression; at the origin, the physical column (`column "QTY" of orders.csv`).
- **What it feeds** — forward to the targets, with the column it ends in (`written to column "TOTAL" of SALES`): the impact analysis of a change.

Lineage reads the same compilation as the run, so it reflects what the engine executes. **Export CSV** writes, for every field written by a target, its source fields and path — a document to attach to a review or a data audit.

07 Filters pushed into database queries

When a **DB Input** sends its rows only to a **Filter** (with the Filter's reject port unconnected), the Filter's conditions are also added to the query as a `WHERE`: the database returns only the rows needed.

Filter condition — text

```
status in ["OPEN", "PAID"] and order_date between "2026-01-01" and "2026-03-31" and qty > 10
```

the condition added to the query (PostgreSQL) — sql

```
WHERE status IN ($1, $2) AND order_date ⩾ $3 AND order_date ⩽ $4 AND qty > $5
-- $1 = 'OPEN', $2 = 'PAID', $3 = '2026-01-01', $4 = '2026-03-31', $5 = 10
```

The result never changes: the Filter still checks every row, and the `WHERE` always keeps at least the rows the Filter keeps. Only comparisons of a field with a known value are pushed (`==` , `!=` , `<` ..., `in` , `between` , `is null` , `like` , `starts_with` ...), with `and` , `or` , `not` ; values travel as query parameters. Calculations on fields (`qty * price > 100`) and functions on fields (`lower(name) == "x"`) stay in the Filter. It applies to PostgreSQL, MySQL/MariaDB and SQLite, not when the query has an `ORDER BY` or the node reads a table with limit/offset. The DB Input's log says what was pushed; its option *Filter pushdown* can switch it off.

08 Built-in functions

107 functions. Arguments in `[…]` are optional. Names are not case-sensitive; aliases are accepted and normalised.

Text (30)

Function	Returns	Aliases	Description
`trim(s)`	string	—	Removes leading and trailing spaces
`ltrim(s)`	string	`trimleft`	Removes leading spaces
`rtrim(s)`	string	`trimright`	Removes trailing spaces
`upper(s)`	string	`touppercase`	Uppercase
`lower(s)`	string	`tolowercase`	Lowercase
`length(s)`	integer	`len`	Number of characters; of a list, number of elements
`substring(s, start [, length])`	string	`substr`	Substring (0-based index)
`replace(s, search, replacement)`	string	—	Replaces all occurrences
`concat(a, b, …)`	string	—	Concatenates
`concat_ws(sep, a, b, …)`	string	—	Concatenates with a separator
`left(s, n)`	string	—	First n characters
`right(s, n)`	string	—	Last n characters
`contains(s, sub)`	boolean	—	True if the text contains sub; contains(list, x) = the list contains x (also: x in list)
`starts_with(s, p)`	boolean	`startswith`	True if it starts with
`ends_with(s, p)`	boolean	`endswith`	True if it ends with
`pad_left(s, n [, char])`	string	`lpad` , `padleft`	Pads on the left up to n
`pad_right(s, n [, char])`	string	`rpad` , `padright`	Pads on the right up to n
`regex_match(s, pattern)`	boolean	`matches`	True if the pattern matches

Function	Returns	Aliases	Description
`like(s, "A%") · s like "A%"`	boolean	—	SQL pattern: % any sequence, _ one character (% _ for the characters themselves). Case-sensitive
`is_empty(x) · x is empty · x is not empty`	boolean	—	True for null, an empty or blank text, or an empty list — the everyday "is there anything?" check. Never null
`split(s, sep [, position])`	depends on arguments	—	Splits into a list; with a position (from 0, negative from the end) returns that part
`index_of(s, sub [, from])`	integer	`indexof`	Position of sub in characters (from 0); -1 if absent
`capitalize(s)`	string	—	First letter uppercase
`title_case(s)`	string	`titlecase`	Each Word Capitalized
`remove_accents(s)`	string	`removeaccents`	Removes accents
`to_slug(s)`	string	`toslug`	text-normalized-for-url
`regex_extract(s, pattern)`	string	`regexextract`	First captured group (or the whole match); null if no match
`replace_regex(s, pattern, repl)`	string	`replaceregex`	Replaces via regular expression
`mask_email(s)`	string	`maskemail`	m****@domain.com
`mask_card(s)`	string	`maskcard`	Shows only the last 4 digits

Numbers (14)

Function	Returns	Aliases	Description
`abs(x)`	number	—	Absolute value
`round(x [, decimals])`	number	—	Rounds
`ceil(x)`	number	—	Rounds up
`floor(x)`	number	—	Rounds down
`sqrt(x)`	number	—	Square root
`power(base, exp)`	number	`pow`	Exponentiation
`min(a, b, …) · min(list)`	depends on arguments	—	The smaller value (null values are skipped); of a list, the smaller element
`max(a, b, …) · max(list)`	depends on arguments	—	The larger value (null values are skipped); of a list, the larger element
`sign(x)`	number	—	-1, 0 or 1 depending on the sign
`negate(x)`	number	—	Changes sign
`clamp(x, min, max)`	depends on arguments	—	Clamps x within the range
`format_number(x, dec [, dec_sep [, thou_sep]])`	string	`formatnumber`	Format: format_number(x,2,",",".") → 1.234,56
`log(x)`	number	`ln`	Natural logarithm
`log10(x)`	number	—	Base-10 logarithm

Dates and times (32)

Function	Returns	Aliases	Description
`now()`	datetime	`current_timestamp`	Current date and time
`today()`	date	`current_date`	Current date
`date_format(d, format)`	string	`formatdate`	Formats a date
`year(d)`	integer	`getyear`	Year
`month(d)`	integer	`getmonth`	Month (1-12)
`day(d)`	integer	`getday`	Day of the month
`hour(d)`	integer	—	Hour (0-23)
`minute(d)`	integer	—	Minutes
`second(d)`	integer	—	Seconds
`quarter(d)`	integer	`getquarter`	Quarter (1-4)
`day_of_week(d)`	integer	`getdayofweek`	Day of week (0=Sunday)
`is_weekend(d)`	boolean	`isweekend`	True if Saturday or Sunday
`add_days(d, n)`	date	`adddays`	Adds n days
`add_months(d, n)`	date	`addmonths`	Adds n months
`add_years(d, n)`	date	`addyears`	Adds n years
`diff_days(a, b)`	integer	`diffdays`	Days from a to b
`add_hours(d, n)`	datetime	—	Adds n hours (date and time)
`add_minutes(d, n)`	datetime	—	Adds n minutes (date and time)
`add_seconds(d, n)`	datetime	—	Adds n seconds (date and time)
`diff_hours(a, b)`	integer	—	Whole hours from a to b
`diff_minutes(a, b)`	integer	—	Whole minutes from a to b
`diff_seconds(a, b)`	integer	—	Seconds from a to b
`to_timezone(d, "Europe/Rome")`	datetime	—	The same instant in another time zone (a date-time without zone is taken as UTC)
`to_utc(d [, "Europe/Rome"])`	datetime	—	The same instant in UTC; the zone says how to read a date-time without zone (default UTC)
`start_of_month(d)`	date	`startofmonth`	First day of the month
`end_of_month(d)`	date	`endofmonth`	Last day of the month
`start_of_year(d)`	date	`startofyear`	First day of the year
`is_before(a, b)`	boolean	`isbefore`	True if a is before b
`is_after(a, b)`	boolean	`isafter`	True if a is after b
`to_unix_timestamp(d)`	integer	`tounixtimestamp`	Seconds since the epoch
`to_unix_timestamp_ms(d)`	integer	`tounixtimestampms`	Milliseconds since the epoch
`parse_date(text [, format])`	datetime	`parsedate`	Parses a text as a date

Conversion, encoding, hashing (12)

Function	Returns	Aliases	Description
`to_string(x)`	string	`str`, `tostring`	Converts to string
`to_int(x)`	integer	`int`, `toint`	Converts to integer
`to_float(x)`	decimal	`float`, `todecimal`	Converts to decimal
`to_bool(x)`	boolean	`bool`, `tobool`	Converts to boolean
`url_encode(s)`	string	`urlencode`	URL-encodes
`url_decode(s)`	string	`urldecode`	URL-decodes
`base64_encode(s)`	string	`base64encode`	Base64-encodes
`base64_decode(s)`	string	`base64decode`	Base64-decodes
`hash_sha256(s)`	string	`hashsha256`	SHA-256 digest (hexadecimal)
`to_json(x)`	string	`tojson`	Serializes to JSON
`hash_sha1(s)`	string	`hashsha1`	SHA-1 (deprecated, compatibility only)
`hash_sha512(s)`	string	`hashsha512`	SHA-512 digest

Logic and null (3)

Function	Returns	Aliases	Description
`coalesce(a, b, …) · a ?? b · try(a, b)`	depends on arguments	`ifnull`, `nvl`, `coalesceempty`, `try`	The first non-null value. A conversion or calculation that fails gives null, so try(to_int(x), 0) falls back to 0
`nullif(a, b)`	depends on arguments	—	null if a == b, otherwise a
`iif(cond, if_true, if_false)`	depends on arguments	`if`	Conditional (equivalent to cond ? a : b)

Lists (10)

Function	Returns	Aliases	Description
`list(a, b, …) · [a, b, …]`	object	—	A list of values
`any(list [, x → condition])`	boolean	—	True if at least one element satisfies the condition (or is true)
`all(list [, x → condition])`	boolean	—	True if every element satisfies the condition (or is true); true for an empty list
`filter(list, x → condition)`	object	—	The elements that satisfy the condition
`map(list, x → expression)`	object	—	Each element transformed
`sum(list [, x → number])`	number	—	Sum of the numeric values (non-numeric and null are skipped)
`count(list [, x → condition])`	integer	—	Elements that satisfy the condition; without it, the non-null elements
`join(list, sep)`	string	—	Joins the elements into a text (null elements are skipped)
`first(list)`	any	—	The first element (null if the list is empty)
`last(list)`	any	—	The last element (null if the list is empty)

Structures (JSON objects and arrays) (6)

Function	Returns	Aliases	Description
`get(object, key)`	any	—	Value of a key
`get_path(object, "a.b.0.c")`	any	`getpath`	Nested value by path
`keys(object)`	object	—	List of keys
`values(object)`	object	—	List of values
`merge(a, b)`	object	—	Merges two objects (b wins)
`decode("table", code [, fallback])`	depends on arguments	—	Looks the code up in a decode table of the project (toolbar → Tables); missing code → the fallback, or null

Lane variables (4)

Function	Returns	Aliases	Description
`var("name")`	any	—	Reads a lane variable
`next("counter")`	integer	—	Advances the counter and returns it: 1, 2, 3… (or from the start value of the variable with the same name). Each node has its own sequence
`current("counter")`	integer	—	Last value given by next(), without advancing; null before the first next()
`reset("counter" [, value])`	any	—	Restarts the counter: the next next() returns the start value, or value

Date formats

`date_format` and `parse_date` accept these patterns (or, for anything else, `strftime` codes such as `%Y-%m-%d`):

Pattern	Meaning	Example for 2026-03-07 14:05:09.250
`yyyy` or `YYYY` / `YY`	year	`2026` / `26`
`MMMM` / `MMM` / `MM`	month name / short name / number	`March` / `Mar` / `03`
`dd` or `DD`	day of the month	`07`
`DDD`	day of the year	`066`
`HH` / `hh`	hour 00–23 / 01–12	`14` / `02`
`mm`	minutes	`05`
`ss`	seconds	`09`
`SSS`	milliseconds	`250`
`A` / `a`	AM/PM / am/pm	`PM` / `pm`

FPEL text

```
date_format(order_date, "dd/MM/yyyy")          // "07/03/2026"
parse_date("07/03/2026", "dd/MM/yyyy")          // 2026-03-07
diff_days(order_date, today()) > 30             // older than 30 days
to_timezone(created_at, "Europe/Rome")          // same instant, Rome time
```

09 Templates

Script templates

The ⚡ **templates** menu of the Script editor inserts these examples. They use only functions the engine has.

Base

Add fields

Computes new fields; the ones you don't touch pass through unchanged.

Script text

```
// Fields are read by name. Assigning creates or overwrites.
// What you do not assign passes downstream as it was.
processed_on  = today()
full_name = concat_ws(" ", name, last_name)
```

Intermediate values with let

Helper calculations that don't end up in the output row.

Script text

```
// "let" does NOT create a field: it only lives inside the script.
let taxable = quantity * unit_price
let vat        = round(taxable * 0.22, 2)

total = round(taxable + vat, 2)
```

Condition

Different branches based on the row content.

Script text

```
if total > 1000 {
  tier = "high"
  log "Order above threshold: " + code
} else if total > 100 {
  tier = "medium"
} else {
  tier = "low"
}
```

Filter (skip)

The rows you don't care about exit through no port.

Script text

```
// "skip" stops processing of THIS row: it exits through no
// port and the following instructions are not executed.
if status ≠ "active" {
  skip
}
```

Reject with a reason (reject)

Sends the row to the reject port explaining why.

Script text

```
// Requires the "reject" port active in the panel. The reason ends up
// in the _reject_reason field of the rejected row.
if email is null {
  reject "email missing"
}
if quantity ⩽ 0 {
  reject "invalid quantity: " + to_string(quantity)
}
```

Fail (error)

Stops the node and sends the error to the lane's error handler.

Script text

```
// Different from reject: here it is the NODE that fails, and the error takes the
// control channel like any other failure.
if record_type is null {
  error "record without type: the file does not have the expected format"
}
```

Data rules (expect)

Declares what the data must respect; the monitor counts the violations of each rule.

Script text

```
// expect <condition> else reject | warn | error ["message"]
//   reject: the row goes to the reject port (needs the port active)
//   warn:   the row continues; the first violation is logged
//   error:  the node fails at the first violation
// At the end of the run every rule reports how many rows broke it.
expect quantity > 0 else reject "quantity must be positive"
expect email like "%@%" else warn "email without @"
expect order_id is not null else error
```

Multiple rows

One row -> N copies

Duplicates each row a number of times.

Script text

```
// "emit" sends downstream a copy of the row as it is at that moment;
// it interrupts nothing. The final "skip" prevents the original
// from exiting TOO: without it, N copies plus the starting row would exit.
repeat quantity as copy {
  copy_number = copy
  emit
}
skip
```

Expand an array

A field containing a JSON array becomes one row per element.

Script text

```
// The field must contain an array (for example from a JSON Parser).
for element in details {
  detail = element
  emit
}
skip
```

Generate rows from nothing

Starting node: no input, it produces the rows itself.

Script text

```
// Set "Row source" to GENERATE: the input port
// disappears, the body runs ONCE and rows exit only
// through "emit". No "skip" needed here: with no input there is
// no original row to hold back.
repeat 12 as month {
  month_number = month
  label       = "month " + to_string(month)
  emit
}
```

Strings

Normalize

Spaces, uppercase, accents.

Script text

```
name    = title_case(trim(name))
code    = upper(trim(code))
search  = to_slug(remove_accents(description))
```

Mask sensitive data

Obfuscated emails and credit cards.

Script text

```
public_email = mask_email(email)
public_card = mask_card(card_number)
```

Extract and replace

Substrings, padding, regular expressions.

Script text

```
prefix    = left(code, 3)
serial      = pad_left(to_string(number), 6, "0")
cleaned     = replace_regex(phone, "[^0-9]", "")
```

Date

Format a date

From date to string in the format you need.

```
Script text
// The pattern accepts both dd/MM/yyyy and %d/%m/%Y.
formatted_date = date_format(order_date, "dd/MM/yyyy")
year_month     = date_format(order_date, "yyyy-MM")
```

Date calculations

Deadlines, differences, quarters.

```
Script text
due_date      = add_days(invoice_date, 30)
days_open     = diff_days(today(), open_date)
quarter_num   = quarter(order_date)

if is_weekend(delivery_date) {
  note = "weekend delivery"
}
```

Numbers

Calculations and rounding

Discounts, totals, values within a range.

```
Script text
let valid_discount = clamp(discount_percent, 0, 100)
let discounted     = price * (1 - valid_discount / 100)

final_price = round(discounted, 2)
savings     = round(price - discounted, 2)
```

Guard against missing values

Default values and safe divisions.

```
Script text
// coalesce returns the first non-null value.
let q = coalesce(quantity, 0)
let t = coalesce(total, 0)

// Division by zero gives null: iif avoids propagating it.
avg_price = iif(q > 0, round(t / q, 2), 0)
```

Checks

Required fields

Rejects incomplete rows saying what is missing.

Script text

```
if code is null {
  reject "code is missing"
}
if description is null {
  reject "description missing for " + code
}
if length(trim(coalesce(description, ""))) < 3 {
  reject "description too short for " + code
}
```

Field format

Checks the shape with a regular expression.

Script text

```
if regex_match(email, "^[^@ ]+@[^@ ]+\\.[a-z]{2,}$") == false {
  reject "invalid email: " + email
}
if starts_with(iban, "IT") == false {
  log "foreign IBAN on " + code
  foreign = true
}
```

Stable key

A reproducible fingerprint from multiple fields.

Script text

```
// concat_ws with a separator prevents "AB"+"C" and "A"+"BC"
// from producing the same key.
key = hash_sha256(concat_ws("|", code, to_string(order_date), customer))
```

Lane variables

Read a lane variable

Values shared in the lane, read with var().

Script text

```
// var("name") reads a variable of the lane (or a shared one of the
// pool). var("name") = … writes it, privately to this node.
environment = var("environment")
load_date   = var("run_date")

if var("environment") == "test" {
  log "row processed in test: " + code
}
```

Inline templates by field type

In the Transform panel (and the field editors of the TMap) each output field can start from a template chosen by the field's type. Besides those below, four presets apply to every type — *Passthrough*, *Null if empty*, *Empty string if null*, *Custom expression…* — and two more to date-times — → *Italian format*, → *Time only (HH:mm)*.

130 templates in all. In the expressions, `$value` is the value of the field and `$param_<name>` a parameter you set in the panel.

Text fields (38)

Template	Expression	What it does
Trim	`trim($value)`	Removes leading and trailing spaces
Trim left	`ltrim($value)`	Removes leading spaces only
Trim right	`rtrim($value)`	Removes trailing spaces only
UPPERCASE	`upper($value)`	Convert to uppercase
lowercase	`lower($value)`	Convert to lowercase
Capitalize	`capitalize($value)`	First letter uppercase
Title Case	`title_case($value)`	First letter of each word uppercase
Slug	`to_slug($value)`	Convert to url-friendly format
Pad left	`pad_left($value, $param_length, $param_char)`	Left-pad with a character
Pad right	`pad_right($value, $param_length, $param_char)`	Right-pad with a character
Substring	`substring($value, $param_start, $param_length)`	Extracts a portion of the string
Replace	`replace($value, $param_from, $param_to)`	Replaces all occurrences
Replace regex	`replace_regex($value, $param_pattern, $param_to)`	Replaces using a regular expression
Empty → null	`iif($value == "" or $value is null, null, $value)`	Returns null if the string is empty
Default if null	`coalesce($value, $param_default)`	Uses a default value if null or empty
Concatenate	`concat($value, $param_sep, $param_suffix)`	Concatenates with another value or string
Join with separator	`concat_ws($param_sep, $value, $param_altro)`	Joins the value with another field or text, using a separator written only once
Add prefix	`concat($param_prefix, "", $value)`	Adds a prefix to the string
Remove accents	`remove_accents($value)`	Normalizes accented characters (è→e, à→a)
Digits only	`replace_regex($value, "[^\\d]", "")`	Removes everything except digits
Letters only	`replace_regex($value, "[^a-zA-Z]", "")`	Removes everything except letters
Length	`length($value)`	Returns the length of the string
→ integer	`to_int($value)`	Converts the string to an integer
→ decimal	`to_float($value)`	Converts the string to a decimal
→ boolean	`to_bool($value)`	Converts true/false/1/0 to a boolean
→ date	`parse_date($value, $param_format)`	Parses a string as a date
Mask email	`mask_email($value)`	Obscures the local part of the email (ab***@domain.com)
Mask card	`mask_card($value)`	Obscures the leading digits of the card (**** **** **** 1234)
Hash SHA-256	`hash_sha256($value)`	Computes the SHA-256 hash of the string
Hash SHA-512	`hash_sha512($value)`	Computes the SHA-512 hash of the string
Hash SHA-1 (deprecated)	`hash_sha1($value)`	Computes SHA-1 hash — only for compatibility with legacy systems

Template	Expression	What it does
Base64 encode	`base64_encode($value)`	Encodes to Base64
Base64 decode	`base64_decode($value)`	Decodes from Base64
URL encode	`url_encode($value)`	Encodes for use in a URL
Contains?	`contains($value, $param_search)`	Checks whether the string contains a substring
Starts with?	`starts_with($value, $param_prefix)`	Checks whether it starts with a prefix
Ends with?	`ends_with($value, $param_suffix)`	Checks whether it ends with a suffix
Matches regex?	`regex_match($value, $param_pattern)`	Tests the string against a regex pattern

Number fields (24)

Template	Expression	What it does
Round	`round($value, $param_decimals)`	Rounds to the specified number of decimals
Floor	`floor($value)`	Rounds down
Ceil	`ceil($value)`	Rounds up
Absolute value	`abs($value)`	Removes the negative sign
Negate	`negate($value)`	Flips the sign
× 100 (percentage)	`$value * 100`	Multiplies by 100 (decimal to percentage)
÷ 100 (from pct)	`$value / 100`	Divides by 100 (percentage to decimal)
Add	`$value + $param_addend`	Adds a fixed value
Multiply	`$value * $param_factor`	Multiplies by a fixed factor
Modulo	`$value % $param_divisor`	Division remainder
Minimum	`max($value, $param_min)`	Clamps the value to the specified minimum
Maximum	`min($value, $param_max)`	Clamps the value to the specified maximum
Clamp	`clamp($value, $param_min, $param_max)`	Clamps the value between min and max
Null → 0	`coalesce($value, 0)`	Returns 0 if the value is null
→ string	`to_string($value)`	Converts the number to a string
EU format	`format_number($value, $param_decimals, ",", ".")`	Formats with European separators (1.234,56)
US format	`format_number($value, $param_decimals, ".", ",")`	Formats with US separators (1,234.56)
Power	`pow($value, $param_exp)`	Raises the number to a power
Square root	`sqrt($value)`	Computes the square root
Natural logarithm	`log($value)`	Computes the natural logarithm
Is positive?	`$value > 0`	Checks whether the number is positive
Is negative?	`$value < 0`	Checks whether the number is negative
Is zero?	`$value == 0`	Checks whether the number is zero
Sign	`sign($value)`	Returns 1, -1 or 0

Integer fields (11)

Template	Expression	What it does
→ string	`to_string($value)`	Converts to a string
Pad zeros	`pad_left(to_string($value), $param_length, "0")`	Formats with zero-padding (e.g. 007)
→ boolean	`$value ≠ 0`	Converts 0/1 to false/true
→ decimal	`to_float($value)`	Converts to a decimal number
Modulo	`$value % $param_n`	Integer division remainder
Null → 0	`coalesce($value, 0)`	Returns 0 if null
Absolute value	`abs($value)`	Removes the sign
Add	`$value + $param_n`	Adds a fixed value
Multiply	`$value * $param_n`	Multiplies by a factor
Is even?	`$value % 2 == 0`	Checks whether the number is even
Is odd?	`$value % 2 ≠ 0`	Checks whether the number is odd

Decimal fields (11)

Template	Expression	What it does
Round 2 dec	`round($value, 2)`	Rounds to 2 decimals
Round 4 dec	`round($value, 4)`	Rounds to 4 decimals
Floor	`floor($value)`	Rounds down
Ceil	`ceil($value)`	Rounds up
Absolute value	`abs($value)`	Removes the negative sign
EU format	`format_number($value, 2, ",", ".")`	1.234,56
US format	`format_number($value, 2, ".", ",")`	1,234.56
× 100	`$value * 100`	Convert from decimal to percentage
→ integer	`to_int($value)`	Truncates the decimals
→ string	`to_string($value)`	Convert to a string
Null → 0.0	`coalesce($value, 0.0)`	Returns 0.0 if null

Boolean fields (8)

Template	Expression	What it does
→ string	`to_string($value)`	"true" / "false"
→ integer	`$value ? 1 : 0`	1 / 0
→ Sì/No (Italian)	`$value ? "Si" : "No"`	"Sì" / "No"
→ Y/N	`$value ? "Yes" : "No"`	"Yes" / "No"
Negate	`!$value`	Flips the boolean value

Template	Expression	What it does
Null → false	`coalesce($value, false)`	Returns false if null
AND	`$value && $param_operand`	Logical AND with a fixed value
OR	`$value || $param_operand`	Logical OR with a fixed value

Date fields (22)

Template	Expression	What it does
Format	`date_format($value, $param_format)`	Formats the date in the chosen format
→ Timestamp unix	`to_unix_timestamp($value)`	Seconds since epoch (1970-01-01)
→ Timestamp ms	`to_unix_timestamp_ms($value)`	Milliseconds since epoch
Year	`year($value)`	Extracts the year (YYYY)
Month	`month($value)`	Extracts the month (1-12)
Day	`day($value)`	Extracts the day of the month (1-31)
Day of week	`day_of_week($value)`	Day of the week (0=Sun, 6=Sat)
Quarter	`quarter($value)`	Quarter (1-4)
Add days	`add_days($value, $param_days)`	Adds N days to the date
Add months	`add_months($value, $param_months)`	Adds N months to the date
Add years	`add_years($value, $param_years)`	Adds N years to the date
Days from today	`diff_days($value, now())`	Difference in days from the current date
Start of month	`start_of_month($value)`	First day of the month of the date
End of month	`end_of_month($value)`	Last day of the month of the date
Start of year	`start_of_year($value)`	First day of the year of the date
Is in the past?	`is_before($value, now())`	Checks whether the date is before today
Is in the future?	`is_after($value, now())`	Checks whether the date is after today
Is weekend?	`is_weekend($value)`	Checks whether the date falls on Saturday or Sunday
Current date	`now()`	Replaces with the current date/time
Today (date only)	`date_format(now(), "YYYY-MM-DD")`	Current date without time
Null → now	`coalesce($value, now())`	Uses the current date if null
→ ISO string	`date_format($value, "YYYY-MM-DD")`	Convert to an ISO 8601 string

Object fields (7)

Template	Expression	What it does
→ JSON string	`to_json($value)`	Serializes the object to a JSON string
Read property	`get($value, $param_key)`	Gets the value of a property
Nested property	`get_path($value, $param_path)`	Gets a nested property by path (e.g. a.b.c)
List keys	`keys($value)`	Array of the object keys

Template	Expression	What it does
List values	`values($value)`	Array of the object values
Merge	`merge($value, $param_extra)`	Merges two objects (the second overrides)
Is null?	`$value is null`	Checks whether the object is null

any (9)

Template	Expression	What it does
→ string	`to_string($value)`	Convert any value to a string
→ integer	`to_int($value)`	Convert to an integer
→ decimal	`to_float($value)`	Convert to a decimal
→ boolean	`to_bool($value)`	Convert to a boolean
Is null?	`$value is null`	Checks whether the value is null
Is not null?	`$value is not null`	Checks whether the value is not null
Coalesce	`coalesce($value, $param_fallback)`	First non-null value of the two
Default if null	`coalesce($value, $param_default)`	Uses a default value if null
If/Else	`$value ? $param_then : $param_else`	Ternary condition