Amtiri

AMTIRI SCRIPT

Syntax

Generated on November 25, 2026

A script is one expression, built from names, literals, operators and function calls. This chapter describes each piece, the order in which operators apply, and the grammar that holds them together. The examples use the sample variables.

Names and case

A name is made of letters, digits and `_`, and does not start with a digit: `LEVEL`, `P101_RUN`, `_last`. Only the letters `a` to `z` and `A` to `Z` count as letters, and a name holds no spaces, dots or dashes, so `X-1` is `X` minus `1`.

- **Case does not matter.** Data Orchester upper-cases every name: variable names, function names and the field names read after a dot. `level`, `Level` and `LEVEL` are the same variable, and `IF` is `if`.
- **Three words are reserved**, in lower case only: `true`, `false` and `null`. `TRUE` is a name like any other, and reading it fails unless a variable has that code.
- **Two constants are preset**: `E` (2.718…) and `PI` (3.14159…). A variable with the code `E` or `PI` replaces them.

```
level # 42.5
Level * 2 # 85.0
IF(PUMP_ON, "on", "off") # "on"
TRUE # error
COUNT-1 # 2.0
PI # 3.141592653589793
```

A name that is not a variable compiles, and fails only when it is read: see Errors and missing values.

Comments

Form	Runs to
`// text`	The end of the line, or of the script
`/** text **/`	The first `**/`, across lines if needed

`/* … */` is not a comment: a single asterisk does not open one. A comment may sit wherever a space may, but a script must hold an expression besides its comments. Inside quotes, `//` and `/**` are plain text.

```
/** a note **/ LEVEL * 2 # 85.0
LEVEL /** a note **/ * 2 # 85.0
LEVEL * 2 // a note # 85.0
/* a note */ LEVEL # syntax error
"http://example.com/a" # "http://example.com/a"
```

Literals

Literal	Written	Gives
Whole number	`42`, `+7`, `007`	A 32-bit whole number, up to `2147483647`. A larger one does not compile
Decimal	`42.5`, `0.5`, `1.5e3`, `2.0E-3`	A decimal. Digits are needed on both sides of the point: `1e3`, `.5` and `5.` do not compile
Text	`"P-101"`	Text, between double quotes
Boolean	`true`, `false`	A boolean, written in lower case
Null	`null`	No value

- **A minus before a number is the unary minus**, never part of the literal. It keeps the kind of number, so `-5` is the whole number -5 and `-2.5` a decimal (see Types and conversions), and it applies to the whole value after it, with its method calls: `-7.3.floor()` is `-(7.3.floor())`, which is `-7.0`. `--5` is `5`.
- **`-2147483648` does not compile**, because the minus applies to `2147483648`, which is too large. `-2147483647 - 1` gives that value as a decimal, `-2147483648.0`, since subtraction always gives a decimal.
- **The point is always a point.** A decimal comma never is one, whatever the plant's language.
- **Nothing else is a literal.** There are no hexadecimal numbers, no digit separators, no single quotes, and no date or colour literals: dates come from date and the other date functions, and colours from rgb and its companions.

Inside text, a backslash starts an escape:

Escape	Gives
`\"`	A double quote
`\\`	A backslash
`\n`, `\r`, `\t`, `\f`	A line break, a carriage return, a tab, a form feed
`\0x` and two hexadecimal digits	The character with that code, from `\0x00` to `\0xFF`: `\0x41` is `A`

Any other escape does not compile. `\0x` always takes exactly two digits, so `"\0x0041"` is the NUL character followed by `41`, not `A`. Text may hold any character, accents and symbols included, written as itself.

```
007 # 7
+5 # 5
-5 # -5
-2.5 # -2.5
-7.3.floor() # -7.0
--5 # 5
2147483647 # 2147483647
2147483648 # syntax error
-2147483648 # syntax error
-2147483647 - 1 # -2.147483648E9
2147483648.0 # 2.147483648E9
1.5e3 # 1500.0
1e3 # syntax error
.5 # syntax error
0x1F # syntax error
"say \"on\"" # "say \"on\""
"line 1\nline 2" # "line 1\nline 2"
"Tank 3 – 45 °C" # "Tank 3 – 45 °C"
"\0x41" # "A"
length("\0x0041") # 3
substr("\0x0041", 1, 3) # "41"
"\0x4" # syntax error
"\q" # syntax error
'P-101' # syntax error
```

Operators and precedence

Operators apply in this order, loosest first. Parentheses group as usual.

Level	Operators	Meaning	Order
1	`||`	Or: `true` when either side is. Stops at the first `true`	Left to right
2	`&&`	And: `true` when both sides are. Stops at the first `false`	Left to right
3	`==` `<>` `!=` `<` `<=` `>` `>=`	Comparisons	Chained in pairs
4	`+` `-`	Add, or join text; subtract	Left to right
5	`*` `/`	Multiply; divide	Left to right
6	`^`	Power	Left to right: `2 ^ 3 ^ 2` is `(2 ^ 3) ^ 2`
7	`-` in front of a value	Negate	—
8	`.field`, `.fn(…)`, `[index]`	Field, method call, item	Left to right

- **Every arithmetic result is a decimal**: `COUNT * 2` is `6.0`. Negation alone keeps the kind: `-COUNT` is `-3`. See Types and conversions.

- `&&` **and** `||` **stop early.** The right side is not evaluated when the left side decides, so it may read something that only exists when the left side holds. The functions and and or take two or more conditions and stop early too.
- **Not in the language**: `%` (use mod), `!` (use not), the words `and` and `or` between values, `? :` (use if), `+=`, `++` and bitwise operators (use bit to test a bit).

`NOPE` below is not a variable, so reading it fails; `||` and `&&` never read it.

```
2 + 3 * 4 # 14.0
(2 + 3) * 4 # 20.0
10 - 2 - 3 # 5.0
100 / 10 / 2 # 5.0
2 ^ 3 ^ 2 # 64.0
-2 ^ 2 # 4.0
-COUNT ^ 2 # 9.0
-COUNT # -3
COUNT * 2 # 6.0
PUMP_ON && COUNT > 2 # true
COUNT > 2 || NOPE # true
false && NOPE # false
NOPE && false # error
10 % 3 # syntax error
mod(10, 3) # 1.0
!PUMP_ON # syntax error
not(PUMP_ON) # false
PUMP_ON and true # syntax error
```

Comparisons

Operator	True when
`==`	Both values are equal
`<>`, `!=`	The values differ. Both spellings are the same operator
`<`, `<=`, `>`, `>=`	The left value sorts before, before or level with, after, after or level with the right one

- **Comparisons chain in pairs.** `1 < COUNT <= 3` means `1 < COUNT && COUNT <= 3`, which reads a range naturally. Each operand is evaluated once. `1 == 1 == true` therefore compares `1` with `true` as well, which is an error.
- **Numbers compare by value**, whatever their kind: `1 == 1.0`. Text compares character by character, and case matters. Dates and times compare in time order.
- **Some pairs cannot be compared.** A number with text is an error, and so is any comparison of lists or records: see Types and conversions.

```
1 < COUNT <= 3 # true
1 < COUNT < 3 # false
SETPOINTS.low < LEVEL < SETPOINTS.high # true
1 == 1.0 # true
TAG == "P-101" # true
TAG == "p-101" # false
TAG <> "P-102" # true
TAG != "P-102" # true
START < START.addHours(1) # true
1 == 1 == true # error
```

Sequences

A script has no statement separator. Several expressions in parentheses, separated by commas, are evaluated in order, and the last one gives the result. One expression in parentheses is plain grouping, and empty parentheses do not compile.

A sequence matters only where an expression is evaluated for what it does, such as an Action button that closes one panel and opens another. The function exec does the same: `exec(a, b)` is `(a, b)`.

```
(1, 2, 3) # 3
(COUNT + 1, "done") # "done"
(COUNT) # 3
() # syntax error
LEVEL; COUNT # syntax error
```

Lists

A list is written `[a, b, c]`, and may mix any kinds of value, lists and records included. `[]` is the empty list, and a comma after the last item does not compile.

`list[i]` reads an item, counting from `0`:

- A decimal index is cut to a whole number: `[1.9]` reads item `1`.
- An index past the end, or below `0`, is an error. length says how many items there are.
- Only lists take an index. Text and records do not.

```
[COUNT + 1, LEVEL > 40, TAG] # [4.0, true, "P-101"]
[] # []
[1, 2,] # syntax error
REGISTERS[0] # 16256
REGISTERS[2] # 65535
REGISTERS[4] # error
[10, 20, 30][1.9] # 20
[[1, 2], [3, 4]][1][0] # 3
REGISTERS.length() # 4
TAG[0] # error
```

Records

A record holds named fields: `{low: 10, high: 90}`. `=` may separate a key from its value instead of `:`, so `{low = 10}` is the same record.

- **Keys are bare names**, never quoted, and are kept as written. A key given twice keeps its last value. `{}` is the empty record, and a comma after the last field does not compile.
- **`.name` reads a field.** The name is matched regardless of case. A field the record does not hold is an error, not `null`: guard it with evl where it may be missing.
- **Records take no index.** `SETPOINTS["low"]` is an error; write `SETPOINTS.low`.
- **Fields have no order.** A record does not keep its fields in the order they were written; see Types and conversions.

```
{low: 10, high = 90} # {high: 90, low: 10}
{a: 1, a: 2} # {a: 2}
{"low": 10} # syntax error
SETPOINTS.low # 10
SETPOINTS.HIGH # 90
SETPOINTS.mid # error
evl(SETPOINTS.mid, 50) # 50
SETPOINTS["low"] # error
{span: SETPOINTS.high - SETPOINTS.low}.span # 80.0
```

Function calls

A function is called by its name and its arguments in parentheses: `round(LEVEL)`, `if(PUMP_ON, "on", "off")`. The function reference lists them all.

- **Checked when compiled.** An unknown function, or the wrong number of arguments, stops the script from compiling.
- **Arguments are handed over unevaluated.** Each function decides whether, and how often, to evaluate each argument. That is how if evaluates only the branch it takes, and how for evaluates its body once per pass.
- **Some arguments are names.** Loops and list functions take the name they bind as a bare name, never in quotes: `forEach(REGISTERS, r, r * 2)`.

```
if(PUMP_ON, "on", NOPE) # "on"
forEach([1, 2, 3], r, r * 10) # [10.0, 20.0, 30.0]
foo(1) # syntax error
if(PUMP_ON) # syntax error
```

Method-call syntax

Any function can be written after its first argument: `x.fn(a, b)` is `fn(x, a, b)`. Calls chain from left to right, which reads well for a value that goes through several steps.

```
TAG.substr(0, 1) # "P"
substr(TAG, 0, 1) # "P"
LEVEL.round() # 43
"abc".to_upper() # "ABC"
[3, 1, 2].arrayMax() # 3
START.addHours(2).year() # 2026
COUNT.to_string().length() # 1
```

Why there is no assignment

A script gives back a value; it never stores one. `=` on its own is not an operator and does not compile, anywhere: not in a formula, not in a dashboard, not in an Action button's script. The palette offers no assignment block.

Every write to a variable goes through the engine, which converts the value to the variable's type, saves it when the variable is persistent, logs it, and recalculates and notifies whatever depends on it. So variables are written this way instead:

To	Use
Calculate a value from others	A variable formula: the engine writes its result
Take whichever of several inputs changed last	A Merger. See Logic modules
Let an operator set a value	A dashboard's input widget bound to the variable: a switch, a button, a field or a slider
Take a value from a device or a peer	The module that reads it writes it

`while` and `dowhile` therefore have nothing to change their condition with: they either never run their body or never stop. Use for, whose counter advances by itself, or a list function such as forEach.

In a record, `=` separates a key from its value and assigns nothing.

```
COUNT = 4 # syntax error
{count = 4} # {count: 4}
```

Compile errors

A script is compiled before it first runs. Compiling checks the syntax, the literals, the function names and the number of arguments. It does not check variable names: a name that is not a variable fails only when the script reads it.

A compile error is reported with the position where compiling stopped. The message starts with `Syntax error at`, shows the text just before and just after that position, marked with `<<==`, and ends with the cause:

The cause reads	When
`Syntax error at [line:column:…]`	The text cannot be read as an expression at that point
`Syntax error at EOF`	The script ends too early, as after a trailing operator
`Process error at [line:column:…]`	The text reads as an expression, but a piece of it cannot be built: a whole number out of range, an unknown escape, an unknown function or a wrong number of arguments

```
Syntax error at
LEVEL *  <<== * 2
Syntax error at [1:9:8/0]
```

```
Syntax error at
date(2026, 9, 29, 14) <<==
Process error at [1:1:0/19]
```

The second script fails because date takes three, five or six arguments.

The formula editor shows this message in its status line; the engine writes it to the log with the variable's code. See Errors and missing values.

Grammar

The grammar in EBNF, as the parser applies it. Blanks, line breaks and comments may stand between any two tokens, except inside text.

```
script      = expression ;
expression  = comparison { ( "||" | "&&" ) comparison } ;
                (* && binds tighter than ||; both stop at the first deciding value *)
comparison  = calculation { compare_op calculation } ;
                (* chained in pairs: a < b < c is a < b && b < c *)
compare_op  = "==" | "<>" | "!=" | "<" | "<=" | ">=" | ">" ;
calculation = operand { ( "+" | "-" | "*" | "/" | "^" ) operand } ;
                (* ^ before * and /, before + and -; each level left to right *)
operand     = indirect | "-" operand ;
                (* the unary minus: -7.3.floor() is -(7.3.floor()) *)
indirect    = value { "." identifier [ "(" [ arguments ] ")" ] | "[" expression "]" } ;
value       = constant | record | list | sequence | identifier | call ;
call        = identifier "(" [ arguments ] ")" ;
arguments   = expression { "," expression } ;
sequence    = "(" expression { "," expression } ")" ;
                (* evaluates each in order and gives the last; one expression groups *)
list        = "[" [ arguments ] "]" ;
record      = "{" [ entry { "," entry } ] "}" ;
entry       = identifier ( ":" | "=" ) expression ;
constant    = integer | decimal | text | "true" | "false" | "null" ;
integer     = [ "+" ] digit { digit } ;
                (* 32-bit; a minus before it is the unary minus *)
decimal     = [ "+" ] digit { digit } "." digit { digit }
              [ ( "e" | "E" ) [ "+" | "-" ] digit { digit } ] ;
text        = '"' { character | escape } '"' ;
character   = ? any character except " and \ ? ;
escape      = "\" ( '"' | "\" | "n" | "r" | "t" | "f" ) | "\0x" hex hex ;
                (* "\0x" takes exactly two hex digits *)
identifier  = ( letter | "_" ) { letter | digit | "_" } ;
                (* except true, false and null *)
letter      = "a" … "z" | "A" … "Z" ;
digit       = "0" … "9" ;
hex         = digit | "a" … "f" | "A" … "F" ;
separator   = { blank | "//" { ? any character except a line break ? }
              | "/**" { ? any character ? } "**/" } ;
```

Next steps

- Types and conversions
- Errors and missing values
- Operators, literals and structures
- The function index

This page describes Amtiri Script 5.4.3 as shipped with Data Orchester engine 6.18.2.