

# json-formula Specification

PDF Association Forms Technical Working Group

0.2.1 : 2023-10-13

## Table of Contents

Abstract .....	3
1. Notation .....	3
2. Data Types .....	3
2.1. Type Coercion .....	4
2.2. Type Coercion Rules .....	4
3. Date and Time Values .....	6
4. Floating Point Precision .....	7
5. Errors .....	7
6. Grammar .....	7
7. Literals .....	10
7.1. JSON Literals .....	10
7.2. String Literals .....	11
7.3. Number literals .....	11
8. Identifiers .....	12
9. Operators .....	13
9.1. Comparison Operators .....	13
9.2. Numeric Operators .....	14
9.3. Concatenation Operator .....	15
9.4. Array Operands .....	15
9.5. Boolean Operators .....	16
9.6. Unary Operators .....	17
10. Expressions .....	18
10.1. Chained Expressions .....	18
10.2. Bracket Expressions .....	19

10.3. Projections .....	22
10.4. Paren Expressions .....	24
10.5. Array Expression .....	24
10.6. Object Expression .....	25
10.7. Wildcard Expressions .....	26
10.8. currentNode .....	27
10.9. Filter Expressions .....	28
10.10. Pipe Expressions .....	30
11. Functions .....	31
11.1. Function Reference .....	33
12. Integrations .....	82
12.1. Globals .....	82
12.2. Specify locale .....	82
12.3. Custom toNumber .....	83
12.4. Additional Functions .....	83
12.5. Hidden Properties .....	83

# Abstract

This document is the specification for json-formula, an expression grammar that operates on JSON documents. The grammar borrows from

- [OpenFormula](#) for spreadsheet operators and function
- [JMESPath](#) for JSON query semantics

## 1. Notation

In the specification, examples are shown through the use of a **search** function. The syntax for this function is:

```
eval(<json-formula expr>, <JSON document>) -> <return value>
```

For simplicity, the json-formula expression and the JSON document are not quoted. For example:

```
eval(foo, {"foo": "bar"}) -> "bar"
```

The result of applying a json-formula expression against a JSON document will result in valid JSON, provided there are no errors during the evaluation process.

## 2. Data Types

json-formula supports all the JSON types:

- number (All numbers are represented as double-precision floating-point)
- string
- boolean (**true** or **false**)
- array (an ordered, sequence of values)
- object (an unordered collection of key value pairs)
- null

There is an additional type that is not a JSON type that's used in json-formula functions:

- expression (A string prefixed with an ampersand (&) character).

## 2.1. Type Coercion

If the supplied data is not correct for the execution context, json-formula will attempt to coerce the data to the correct type. Coercion will occur in these contexts:

- Operands of the concatenation operator (&) shall be coerced to a string, except when an operand is an array. Arrays shall be coerced to an array of strings.
- Operands of numeric operators (+, -, \*, /) shall be coerced to numbers except when the operand is an array. Arrays shall be coerced to an array of numbers.
- Operands of the union operator (~) shall be coerced to an array
- The left-hand operand of ordering comparison operators (>, >=, <, <=) must be a string or number. Any other type shall be coerced to a number.
- If the operands of an ordering comparison are different, they shall both be coerced to a number
- Parameters to functions shall be coerced to the expected type as defined by the function signature
- Slice and index operations shall coerce operands to a number

The equality and inequality operators (=, ==, !=, <>) do **not** perform type coercion. If operands are different types, the values are considered not equal.

Coercion is not always possible, and if so, an **invalid-type error** shall be emitted.

### Examples

```
eval("abc" & 123, {}) -> "abc123"
eval("123" * 2, {}) -> 246
eval([1,2,3] ~ 4, {}) -> [1,2,3,4]
eval(123 < "124", {}) -> true
eval("23" > 111, {}) -> false
eval(abs("-2"), {}) -> 2
eval([1,2,3,4]["1"], {}) -> 2
eval(1 == "1", {}) -> false
```

## 2.2. Type Coercion Rules

Provided Type	Expected Type	Result
number	string	number converted to a string. Similar to JavaScript <code>toString()</code>
boolean	string	"true" or "false"

Provided Type	Expected Type	Result
array	string	Not supported
object	string	Not supported
null	string	"" (empty string)
string	number	Parse string to a number. If the string is not a well-formed number, will return 0
boolean	number	true → 1 false → 0
array	number	Not supported
object	number	Not supported
null	number	null
number	array	create a single-element array with the number
string	array	create a single-element array with the string
boolean	array	create a single-element array with the boolean
object	array	Not supported
null	array	Empty array
number	object	No supported
string	object	Not supported
boolean	object	Not supported
array	object	Not supported. Use: <code>fromEntries(entries(array))</code>
null	object	Empty object
number	boolean	zero is false, all other numbers are true
string	boolean	Empty string is false, populated strings are true
array	boolean	Empty array is false, populated arrays are true

Provided Type	Expected Type	Result
object	boolean	Empty object is false, populated objects are true
null	boolean	false

## Examples

```
eval("\$123.00\" + 1", {}) -> 1"
eval("truth is " & `true`, {}) -> "truth is true"
eval(2 + `true`, {}) -> 3
eval(avg("20"), {}) -> 20
```

## 3. Date and Time Values

In order to support date and time functions, json-formula needs to represent date and time values. Date/time values are represented as a number where:

- The integral portion of the number represents the number of days since the epoch: January 1, 1970, [UTC](#)
- The fractional portion of the number represents the fractional portion of the day
- The date/time value is offset from the current time zone to UTC
- The current time zone is determined by the host operating system

The preferred ways to create a date/time value are by using one of these functions:

- `datetime()`
- `now()`
- `today()`
- `time()`

## Examples

```
eval(datetime(1970,1,2,0,0,0) - datetime(1970,1,1,0,0,0), {}) -> 1
eval(datetime(2010,1,21,12,0,0) |
  {month: month(@), day: day(@), hour: hour(@)}, {}) ->
  {"month": 1, "day": 21, "hour": 12}
```

## 4. Floating Point Precision

Numbers are represented in [double-precision floating-point format](#). As with any system that uses this level of precision, results of expressions may be off by a tiny fraction. e.g.  $10 * 1.44 \rightarrow 14.399999999999999$

Authors should mitigate this behavior:

- When comparing fractional results, do not compare for exact equality. Instead, compare within a range. e.g.: instead of: `a == b`, use: `abs(a-b) < 0.000001`
- leverage the built-in functions that manipulate fractional values:
  - `ceil()`
  - `floor()`
  - `round()`
  - `trunc()`

## 5. Errors

Errors may be raised during the json-formula evaluation process. How and when errors are raised is implementation specific, but implementations should indicate to the caller when errors have occurred.

The following errors are defined:

- `invalid-type` is raised when an invalid type is encountered during the evaluation process.
- `invalid-value` is raised when an invalid value is encountered during the evaluation process.
- `unknown-function` is raised when an unknown function is encountered during the evaluation process.
- `invalid-arity` is raised when an invalid number of function arguments is encountered during the evaluation process.

## 6. Grammar

The grammar is specified using [Antlr](#).

```
grammar JsonFormula;
```

```

formula : expression EOF ;

expression
: expression '.' chainedExpression
| expression bracketExpression
| bracketExpression
| expression ('*' | '/' | '&'amp;' | '~') expression
| expression ('+' | '-') expression
| expression COMPARATOR expression
| expression '&&' expression
| expression '||' expression
| identifier
| '!' expression
| '-' expression
| '(' expression ')'
| wildcard
| arrayExpression
| objectExpression
| JSON_FRAGMENT
| functionExpression
| expression '|' expression
| STRING
| (REAL_OR_EXPONENT_NUMBER | INT)
| currentNode
;

chainedExpression
: identifier
| arrayExpression
| objectExpression
| functionExpression
| wildcard
;

wildcard : '*' ;

arrayExpression : '[' expression (',' expression)* ']' ;

objectExpression
: '{' '}'
| '{' keyvalExpr (',' keyvalExpr)* '}'
;

keyvalExpr : identifier ':' expression ;

bracketExpression
: '[' '*' ']'
| '[' slice ']'
| '[' ']'
| '['? expression ']'
| '[' expression ']'
;

slice : start=expression? ':' stop=expression? (':' step=expression?)? ;

```



```

COMPARATOR
: '<'
| '<='
| '=='
| '='
| '>='
| '>'
| '!='
| '<>'
;

functionExpression
: NAME '(' functionArg (',' functionArg)* ')'
| NAME '(' ')'
;

functionArg
: expression
| expressionType
;

currentNode : '@' ;

expressionType : '&' expression ;

identifier
: NAME
| QUOTED_NAME
;

NAME : [a-zA-Z_$] [a-zA-Z0-9_$]* ;

QUOTED_NAME : '\\' (ESC | ~ ['\\'])* '\\';

JSON_FRAGMENT
: '\\' (STRING | ~ [\\`]+)* '\\'
;

STRING : '"' (ESC | ~ [ "\\`]) * '"' ;

fragment ESC : '\\' (UNICODE | [bfnr\\`'"/]);

fragment UNICODE
: 'u' HEX HEX HEX HEX
;

fragment HEX
: [0-9a-fA-F]
;

REAL_OR_EXPONENT_NUMBER
: INT? '.' [0-9] + EXP?
| INT EXP
;

```

```

INT
: [0-9]+
;

fragment EXP
: [Ee] [+|-]? INT
;

WS
: [ \t\n\r] + -> skip
;

```

In addition to the grammar, there is a token precedence ordered from weakest to tightest binding:

- pipe: `|`
- or: `||`
- and: `&&`
- concatenate: `&`
- add: `+`, subtract: `-`
- multiply: `*`, divide: `/`, union: `~`
- Equals: `=` (or `==`), Greater than: `>`, Less than: `<`, Greater than or equal: `>=`, Less than or equal: `<=`, Not equals: `!=` (or `<>`)
- Flatten: `[]`
- Unary not `!`, unary minus: `-`
- Dot `.` (expression combinations)
- Braces `{}`
- Bracket Expression `[...]`
- Parenthesis `()`

## 7. Literals

### 7.1. JSON Literals

```
jsonLiteral = `` jsonValue ``
```

A JSON literal expression allows arbitrary JSON objects to be specified. This is allowed anywhere an expression is permitted. Implementations should use an existing JSON parser to parse these literals.

Note that the backtick character ( ``` ) character must now be escaped in a JSON literal which means implementations need to handle this case before passing the resulting string to a JSON parser.

## Examples

```
eval(`"foo"`, {}) -> "foo"
eval(`"foo\`bar"`, {}) -> "foo`bar"
eval(`[1, 2]`, {}) -> [1, 2]
eval(`true`, {}) -> true
eval(`{"a": "b"}`.a, {}) -> "b"
eval({first: a, type: `mytype`}, {"a": "b", "c": "d"})
  -> {"first": "b", "type": "mytype"}
```

## 7.2. String Literals

```
STRING : '"' (ESC | ~["\\])* '"' ;

fragment ESC : '\\\'' (UNICODE | [bfnr\t\\\'"/]);

fragment UNICODE
  : 'u' HEX HEX HEX HEX
  ;

fragment HEX
  : [0-9a-fA-F]
  ;
```

A **STRING** literal is a value enclosed in double quotes and supports the same character escape sequences as strings in JSON. e.g., a character 'A' could be specified as the unicode sequence: `\u0041`.

A string literal can also be expressed as a JSON literal. For example, the following expressions both evaluate to the same value: "foo"

```
eval(`"foo"`, {}) -> "foo"
eval("foo", {}) -> "foo"
```

## 7.3. Number literals

```

numberLiteral = REAL_OR_EXPONENT_NUMBER | INT

REAL_OR_EXPONENT_NUMBER
: INT? '.' [0-9] + EXP?
| INT EXP
;

INT
: [0-9]+
;

fragment EXP
: [Ee] [+\\-]? INT
;

```

Number literals follow the same syntax rules as numeric values in JSON with two exceptions:

1. Number literals may omit a leading zero. For example, **.123** is not valid JSON, but is allowed as a number literal.
2. The grammar construction for a number literal does not include a minus sign. Literal expressions are made negative by prefixing them with a unary minus.

Note that number literals (and JSON numbers) allow scientific notation.

Allowing number literals in expressions leads to an ambiguity: a bracket with a single signed digit e.g.: **[0]** can be interpreted as a flatten operation or an **arrayExpression** with the number zero. To resolve this ambiguity, the grammar sets a precedence order so that **[-?[0-9]]** is treated as an **index expression**. To explicitly express an array with one element, use a JSON literal: **`[0]`**

## Examples

```

eval(44, {}) -> 44
eval([12, 13], {}) -> [12, 13]
eval({a: 12, b: 13}, {}) -> {"a": 12, "b": 13}
eval(foo | [1], {"foo": [3,4,5]}) -> 4
eval(foo | @[-1], {"foo": [3,4,5]}) -> 5
eval(foo | [1, 2], {"foo": [3,4,5]}) -> [1, 2]
eval(6 / 3, {}) -> 2
eval(1e2, {}) -> 100

```

## 8. Identifiers

```
identifier
: NAME
| QUOTED_NAME
;

NAME : [a-zA-Z_$] [a-zA-Z0-9_$]* ;

QUOTED_NAME : '\'' (ESC | ~ ['\\])* '\'';
```

An **identifier** is the most basic expression and can be used to extract a single element from a JSON document. The return value for an **identifier** is the value associated with the identifier. If the **identifier** does not exist in the JSON document, then a **null** value is returned.

From the grammar rule listed above identifiers can be one or more characters, and must start with a character in the range: **A-Za-z\_\$**.

When an identifier has a character sequence that does not match the **NAME** grammar rule, it must be enclosed in single quotes. In this situation, an identifier is specified with a single quote (**'**), followed by any number of characters, followed by another single quote. Any valid string can be used between single quotes, include JSON supported escape sequences.

## Examples

```
eval(foo, {"foo": "value"}) -> "value"
eval(bar, {"foo": "value"}) -> null
eval(foo, {"foo": [0, 1, 2]}) -> [0, 1, 2]
eval('with space', {"with space": "value"}) -> "value"
eval('special chars: !@#"', {"special chars: !@#": "value"}) -> "value"
eval('quote\'char', {"quote'char": "value"}) -> "value"
eval('\u2713', {"\u2713": "value"}) -> "value"
```

## 9. Operators

### 9.1. Comparison Operators

The following comparison operators are supported:

- **=**, **==**: test for equality
- **!=**, **<>**: test for inequality
- **<**: less than

- `<=`: less than or equal to
- `>`: greater than
- `>=`: greater than or equal to

### 9.1.1. Equality Operators

Two representations of the equality and inequality operators are supported: `=` and `==` are equivalent in functionality. Both variations are supported in order to provide familiarity to users with experience with similar grammars. Similarly, `!=` and `<>` function identically. Note that there is no ambiguity with the `=` operator, since json-formula does not have an assignment operator.

- A `string` is equal to another `string` if they have the exact same sequence of code points
- `number` values are compared for an exact match. When comparing fractional values, authors should take into account `floating point precision` considerations.
- The literal values `true/false/null` are equal only to their own literal values
- Two JSON objects are equal if they have the same set of keys and values. Two JSON objects `x` and `y`, are considered equal if they have the same number of key/value pairs and if, for each key value pair `(i, j)` in `x`, there exists an equivalent pair `(i, j)` in `y`
- Two JSON arrays are equal if they have equal elements in the same order. Two arrays `x` and `y` are considered equal if they have the same length and, for each `i` from `0` until `length(x)`, `x[i] == y[i]`

### 9.1.2. Ordering Operators

The comparisons follow these rules:

- If both operands are numbers, a numeric comparison is performed
- If both operands are strings, they are compared as strings, based on the values of the Unicode code points they contain
- If operands are mixed types, `type coercion` is applied before performing the comparison

## 9.2. Numeric Operators

The following operators work with numeric operands:

- addition: `+`
- subtraction: `-`

- multiplication  $\star$
- division:  $/$

```
eval(left + right, {"left": 8, "right": 12 }) -> 20
eval(right - left - 10, {"left": 8, "right": 12 }) -> -6
eval(4 + 2 * 4, {}) -> 12
eval(10 / 2 * 3, {}) -> 15
```

## 9.3. Concatenation Operator

The concatenation operator ( $\&$ ) takes two string operands and combines them to form a single string.

```
eval(left & value & right,
  {"left": "[", "right": "]", "value": "abc" }) -> "[abc]"
eval(map("&$$" & @, values), {"values": [123.45, 44.32, 99.00] }) ->
  ["$123.45", "$44.32", "$99"]
```

## 9.4. Array Operands

The numeric and concatenation operators ( $+$ ,  $-$ ,  $\star$ ,  $/$ ,  $\&$ ) have special behavior when applied to arrays.

- When both operands are arrays, a new array is returned where the elements are populated by applying the operator on each element of the left operand array with the corresponding element from the right operand array
- If both operands are arrays and they do not have the same size, the shorter array is padded with null values
- If one operand is an array and one is a scalar value, a new array is returned where the operator is applied with the scalar against each element in the array

```
eval([1,2,3] + [2,3,4], {}) -> [3,5,7]
eval([1,2,3,4] * [1,2,3], {}) -> [1,4,9,0]
eval([1,2,3,4] & "%", {}) -> ["1%", "2%", "3%", "4%"]
```

### 9.4.1. Union Operator

The union operator ( $\sim$ ) returns an array formed by concatenating the contents of two arrays.

```
eval(a ~ b, {"a": [0,1,2], "b": [3,4,5]}) -> [0,1,2,3,4,5]
eval(a ~ b, {"a": [[0,1,2]], "b": [[3,4,5]]}) -> [[0,1,2],[3,4,5]]
eval(a[] ~ b[], {"a": [[0,1,2]], "b": [[3,4,5]]}) -> [0,1,2,3,4,5]
eval(a ~ 10, {"a": [0,1,2]}) -> [0,1,2,10]
eval(a ~ `null`, {"a": [0,1,2]}) -> [0,1,2]
```

## 9.5. Boolean Operators

### 9.5.1. Or Operator

The OR operator (`||`) will evaluate to either the left operand or the right operand. If the left operand can be coerced to a true value, it is used as the return value. If the left operand cannot be coerced to a true value, then the right operand is used as the return value.

The following conditions cannot be coerced to true:

- Empty array: `[]`
- Empty object: `{}`
- Empty string: `""`
- False boolean: `false`
- Null value: `null`
- zero value: `0`

### Examples

```
eval(foo || bar, {"foo": "foo-value"}) -> "foo-value"
eval(foo || bar, {"bar": "bar-value"}) -> "bar-value"
eval(foo || bar, {"foo": "foo-value", "bar": "bar-value"}) -> "foo-value"
eval(foo || bar, {"baz": "baz-value"}) -> null
eval(foo || bar || baz, {"baz": "baz-value"}) -> "baz-value"
eval(override || myarray[-1], {"myarray": ["one", "two"]}) -> "two"
eval(override || myarray[-1], {"myarray": ["one", "two"], "override": "yes"})
  -> "yes"
```

### 9.5.2. And Operator

The AND operator (`&&`) will evaluate to either the left operand or the right operand. If the left operand is a truth-like value, then the right operand is returned. Otherwise the left operand is returned. This reduces to the expected truth table:



LHS	RHS	Result
True	True	True
True	False	False
False	True	False
False	False	False

This is the standard truth table for a [logical conjunction](#).

### Examples

```
eval(True && False, {"True": true, "False": false}) -> false
eval(Number && EmptyList, {"Number": 5, "EmptyList": []}) -> []
eval(foo[?a == `1` && b == `2`],
    {"foo": [{"a": 1, "b": 2}, {"a": 1, "b": 3}]} -> [{"a": 1, "b": 2}]
```

## 9.6. Unary Operators

### 9.6.1. Not Operator

A unary NOT operator (!) is a boolean operator that negates the result of an expression. If the expression results in a truth-like value, NOT operator will change this value to **false**. If the expression results in a false-like value, a NOT operator will change the value to **true**.

### Examples

```
eval(!True, {"True": true}) -> false
eval(!False, {"False": false}) -> true
eval(!Number, {"Number": 5}) -> false
eval(!EmptyList, {"EmptyList": []}) -> true
```

### 9.6.2. Minus Operator

A unary Minus operator (-) is a numeric operator that negates the value of an operand.

### Examples

```
eval(-11, {}) -> -11
eval(-n, {"n": 5, "nn": -10}) -> -5
eval(-nn, {"n": 5, "nn": -10}) -> 10
eval(--n, {"n": 5, "nn": -10}) -> 5
```

## 10. Expressions

### 10.1. Chained Expressions

```
expression: expression '.' chainedExpression

chainedExpression
: identifier
| arrayExpression
| objectExpression
| functionExpression
| wildcard
;

wildcard : '*' ;
```

A chained expression is a combination of two expressions separated by the dot (.) char. A chained expression is evaluated as follows:

- Evaluate the expression on the left against the source JSON document
- Evaluate the expression on the right against the result of the left expression evaluation

In pseudo-code

```
left-evaluation = eval(left-expression, original-json-document)
result = eval(right-expression, left-evaluation)
```

A chained expression is itself an expression, so there can be multiple levels of chained expressions: **grandparent.parent.child**.

#### Examples

Given a JSON document: **{"foo": {"bar": "baz"}}**, and a json-formula expression: **foo.bar**, the evaluation process would be

```
left-evaluation = eval(foo, {"foo": {"bar": "baz"}}) -> {"bar": "baz"}
result = eval(bar, {"bar": "baz"}) -> "baz"
```

The final result in this example is **"baz"**.

Additional examples

```
eval(foo.bar, {"foo": {"bar": "value"}}) -> "value"
eval(foo.'bar', {"foo": {"bar": "value"}}) -> "value"
eval(foo.bar, {"foo": {"baz": "value"}}) -> null
eval(foo.bar.baz, {"foo": {"bar": {"baz": "value"}}}) -> "value"
```

## 10.2. Bracket Expressions

```
expression: expression bracketExpression
```

```
bracketExpression
: '[' '*' ']'
| '['? expression ']'
| '[' expression ']'
| '[' slice ']'
| '[' ']'
;
```

From the **bracketExpression** construction, the bracketed contents provide access to the elements in an array or object.

- The wildcard: (**'[' ' \* ' ]'**) variation is discussed in the [Wildcard Expressions](#) section
- The filtering: (**'[? ' expression ' ]'**) variation is discussed in the [Filter Expressions](#) section

### 10.2.1. Index Expressions

When brackets enclose an expression (**'[ ' expression ' ]'**), the results of that expression are used to index into an object or array.

When used with an object, the bracketed expression is used as the name of a key in that object. For example, **foo["bar"]** is equivalent to **foo.bar**. If the expression returns a key not present in the object, a **null** value is returned.

Most commonly, index expressions are used with arrays. With arrays, an expression in brackets is expected to return an integer value to return an offset in the array. Indexing is 0 based where an index of 0 refers to the first element of the array. A negative index indicates that indexing is relative to the

end of the array, specifically:

```
negative-index == (length of array) + negative-index
```

Given an array of length **N**, an index of **-1** would be equal to a positive index of **N - 1**, which is the last element of the array. An expression is outside the bounds of the array when the value is greater than or equal to the length of the array or less than the negative length of the array. If an index expression is outside the bounds of the array then a value of **null** is returned.

Index expressions are evaluated in the context of the left hand expression. For example, given the source JSON:

```
{
  "partners": {
    "partner1": "Joe",
    "partner2": "Frank",
    "seniorPartner": "partner1"
  }
}
```

The expression: **partners[@.seniorPartner]** (or **partners[seniorPartner]**) will evaluate to **"Joe"**. Specifically, the nested expression inside: **partners[partners.seniorPartner]** will return **null**.

## Examples

```
eval(foo["bar"], {"foo": {"bar": 21}}) -> 21
eval(a[1], {a: [5,6,7,8,9]}) -> 6
eval(a[-2], {a: [5,6,7,8,9]}) -> 8
eval([0], ["first", "second", "third"]) -> "first"
eval([-1], ["first", "second", "third"]) -> "third"
eval([100], ["first", "second", "third"]) -> null
eval(foo[0], {"foo": ["first", "second", "third"]}) -> "first"
eval(foo[100], {"foo": ["first", "second", "third"]}) -> null
eval(foo[0][0], {"foo": [[0, 1], [1, 2]]}) -> 0
```

## 10.2.2. Slices

```
slice : start=expression? ':' stop=expression? (':' step=expression?)? ;
```

A slice expression allows you to select a contiguous subset of an array. A slice has a **start**, **stop**, and **step** value. The general form of a slice is **[start:stop:step]**. Each component of the slice is

optional and can be omitted, but there must be at least one colon (:) character.

Slices in json-formula have the same semantics as python slices. If you're familiar with python slices, you're familiar with json-formula slices.

Given a **start**, **stop**, and **step** value, the sub elements in an array are extracted as follows:

- The first element in the extracted array is the index denoted by **start**.
- The last element in the extracted array is the index denoted by **end - 1**.
- The **step** value determines the amount by which the index increases or decreases. The default step value is 1. For example, a step value of 2 will return every second value from the array. If step is negative, slicing is performed in reverse—from the last (stop) element to the start.

Slice expressions adhere to the following rules:

- If a negative start position is given, it is calculated as the total length of the array plus the given start position.
- If no start position is given, it is assumed to be 0 if the given step is greater than 0 or the end of the array if the given step is less than 0.
- If a negative stop position is given, it is calculated as the total length of the array plus the given stop position.
- If no stop position is given, it is assumed to be the length of the array if the given step is greater than 0 or 0 if the given step is less than 0.
- If the given step is omitted, it is assumed to be 1.
- If the given step is 0, an **invalid-value** error must be raised.
- If the element being sliced is not an array, the result is **null**.
- If the element being sliced is an array and yields no results, the result must be an empty array.

## Examples

```
eval([0:4:1], [0, 1, 2, 3]) -> [0, 1, 2, 3]
eval([0:4], [0, 1, 2, 3]) -> [0, 1, 2, 3]
eval([0:3], [0, 1, 2, 3]) -> [0, 1, 2]
eval([:2], [0, 1, 2, 3]) -> [0, 1]
eval([::2], [0, 1, 2, 3]) -> [0, 2]
eval([::-1], [0, 1, 2, 3]) -> [3, 2, 1, 0]
eval([-2:], [0, 1, 2, 3]) -> [2, 3]
```

### 10.2.3. Flatten Operator

When the character sequence `[]` is provided as a bracket specifier, then a flattening operation occurs on the current result. The flattening operator will merge one level of sub-arrays in the current result into a single array. The flattening operator has the following semantics:

- Create an empty result array
- Iterate over the elements of the current result
- If the current element is not an array, add to the end of the result array
- If the current element is an array, add each element of the current element to the end of the result array
- The result array is returned as a [projection](#)

Once the flattening operation has been performed, subsequent operations are projected onto the flattened array. The difference between a bracketed wildcard (`[*]`) and flatten (`[]`) is that flatten will first merge sub-arrays.

### Examples

```
eval(foo[], {"foo": [[0, 1], [1, 2], 3]}) -> [0,1,1,2,3]
eval(foo[], {"foo": [[0, 1], [1, 2], [3,[4,5]]]}) -> [0,1,1,2,3,[4,5]]
eval(foo[][], {"foo": [[0, 1], [1, 2], [3,[4,5]]]}) -> [0,1,1,2,3,4,5]
```

## 10.3. Projections

Projections allow you to apply an expression to a collection of elements. Projections are created when any form of a [Bracket Expression](#) transforms a source array or when a [wildcard](#) is applied to an object:

Given the source JSON:

```
{
  "items": [
    {
      "desc": "pens",
      "price": 3.23
    },
    {
      "desc": "pencils",
      "price": 1.34
    },
    {
      "desc": "staplers",
      "price": 10.79
    }
  ]
}
```

These expressions will create a projection:

- `items[*]`
- `items[]`
- `items[0:2]`
- `items[?expr]`
- `items[0].*`

The behavior of [Chained Expressions](#) and [Bracket Expressions](#) are changed to operate on each element of the projection, rather than the array itself.

## Examples

```
eval(items[*].desc, items) -> ["pens", "pencils", "staplers"]
eval(items[*].desc.upper(@), items) -> ["PENS", "PENCILS", "STAPLERS"]
eval(items[0].*, items) -> [
  ["pens", 3.23],
  ["pencils", 1.34],
  ["staplers", 10.79]
]
eval(items[0:2].price * 2, items) -> [6.46, 2.68]
eval(items[?price < 3], items) -> [{"desc": "pencils", "price": 1.34}]
```

A [pipe expression](#) will stop the current projection and process it as a normal array.

For example, if you wanted to sum the `price` values, this expression: `items[*].price.sum(@)` will sum each individual price, returning: `[3.23, 1.34, 10.79]`. Whereas using a pipe operator results in: `items[*].price | sum(@) -> 15.36`.

## 10.4. Paren Expressions

```
parenExpression = '(' expression ')'
```

A **parenExpression** allows a user to override the precedence order of an expression  
e.g. **(a || b) && c**

### Examples

```
eval(foo[?(a == 1 || b == 2) && c == 5],  
      {"foo": [{"a": 1, "b": 2, "c": 3}, {"a": 3, "b": 4}]} -> []
```

## 10.5. Array Expression

```
arrayExpression : '[' expression (',' expression)* ']' ;
```

An array expression is used to extract a subset of elements from the JSON document into an array. Within the start and closing brackets are one or more expressions separated by a comma. Each expression will be evaluated against the JSON document, and each result will be added to the array

An **arrayExpression** with **N** expressions will result in an array of length **N**. Given an array expression **[expr-1,expr-2,...,expr-n]**, the evaluated expression will return **[evaluate(expr-1), evaluate(expr-2), ..., evaluate(expr-n)]**.

Given an array expression: **[n, "doubled", n \* 2]** and the data **{"n": 4}**, the expression is evaluated as follows:

1. An empty array is created: **[]**
2. The expression **n** is evaluated against the source document and the result (**4**) is appended to the array
3. The literal expression **"doubled"** is appended to the array
4. The expression **n \* 2** is evaluated against the source document and the result (**8**) is appended to the array

The final result will be: **[4, "doubled", 8]**.



## Examples

```
eval([foo,bar], {"foo": "a", "bar": "b", "baz": "c"}) -> ["a", "b"]
eval([foo,bar[0]], {"foo": "a", "bar": ["b"], "baz": "c"}) -> ["a", "b"]
eval([foo,bar.baz], {"foo": "a", "bar": {"baz": "b"}}) -> ["a", "b"]
eval([foo,baz], {"foo": "a", "bar": "b"}) -> ["a", null]
```

## 10.6. Object Expression

```
objectExpression = "{" ( keyvalExpr ( "," keyvalExpr )*)? "}"
keyvalExpr = identifier ":" expression
```

An object expression is used to extract a subset of elements from the JSON document into an object. An **objectExpression** requires key names to be provided, as specified in the **keyvalExpr** rule. Given the following rule

```
keyvalExpr = identifier ":" expression
```

The **identifier** is used as the key name and the result of evaluating the **expression** is the value associated with the **identifier** key.

Each **keyvalExpr** within the **objectExpression** will correspond to a single key value pair in the created object. Unlike the **arrayExpression**, an **objectExpression** may be empty.

## Examples

Given an object expression **{foo: one.two, bar: bar}** and the data **{"bar": "bar", "one": {"two": "one-two"}}**, the expression is evaluated as follows:

1. An object is created: **{}**
2. A key **foo** is created whose value is the result of evaluating **one.two** against the provided JSON document: **{"foo": evaluate(one.two, <data>)}**
3. A key **bar** is created whose value is the result of evaluating the expression **bar** against the provided JSON document.

The final result will be: **{"foo": "one-two", "bar": "bar"}**.

Additional examples:

```
eval({foo: foo, bar: bar}, {"foo": "a", "bar": "b", "baz": "c"})
  -> {"foo": "a", "bar": "b"}
eval({foo: foo, firstbar: bar[0]}, {"foo": "a", "bar": ["b"]})
  -> {"foo": "a", "firstbar": "b"}
eval({foo: foo, 'bar.baz': bar.baz}, {"foo": "a", "bar": {"baz": "b"}})
  -> {"foo": "a", "bar.baz": "b"}
eval({foo: foo, baz: baz}, {"foo": "a", "bar": "b"})
  -> {"foo": "a", "baz": null}
```

## 10.7. Wildcard Expressions

There are two forms of wildcard expression:

1. `[*]` from the `bracketExpression` construction:

```
bracketExpression
: '[' '*' ']'
| '[' slice ']'
| '[' ']'
| '['? expression ']'
| '[' expression ']'
;
```

2. `.*` from the `chainedExpression` construction:

```
expression : expression '.' chainedExpression

chainedExpression
: identifier
| arrayExpression
| objectExpression
| functionExpression
| wildcard
;

wildcard : '*' ;
```

The `[*]` syntax (referred to as an array wildcard expression) is valid for array expressions. The array wildcard will return a [projection](#) with all the elements of the source array. If an array wildcard expression is applied to any other JSON type, a value of `null` is returned.

The `.*` syntax (referred to as an object wildcard expression) is valid for object expressions. The object wildcard will return an array of the values from the object key/value pairs. If an object wildcard expression is applied to any other JSON type, a value of `null` is returned.

Note that JSON objects are explicitly defined as unordered. Therefore an object wildcard expression can return the values associated with the object in any order.

## Examples

```
eval([*].foo, [{"foo": 1}, {"foo": 2}, {"foo": 3}]) -> [1, 2, 3]
eval([*].foo, [{"foo": 1}, {"foo": 2}, {"bar": 3}]) -> [1, 2, null]
eval(*.foo, {"a": {"foo": 1}, "b": {"foo": 2}, "c": {"bar": 1}}) ->
[1, 2, null]
```

## 10.8. currentNode

```
currentNode : '@' ;
```

The **currentNode** token represents the node being evaluated in the current context. The **currentNode** token is commonly used for:

- Functions that require the current node as an argument
- [Filter Expressions](#) that examine elements of an array
- Access to the current context in [projections](#).

json-formula assumes that all expressions operate on the current node. Because of this, an expression such as **@.name** would be equivalent to just **name**.

### 10.8.1. currentNode state

At the start of an expression, the value of the current node is the data being evaluated by the json-formula expression. As an expression is evaluated, **currentNode** must change to reflect the node being evaluated. When in a [projection](#), the current node value must be changed to the node being evaluated by the projection.

## Examples

Given:

```
{
  "family": [
    {"name": "frank", "age": 22},
    {"name": "jane", "age": 23}
  ]
}
```

`eval(@.family[0].name, {...}) -> "frank"`

`eval(family[][left(@.name), age], {...}) -> [{"f", 22}, {"j", 23}]`

`eval(family[?@.age == 23], {...}) -> [{"name": "jane", "age": 23}]`

`eval(family[?age == 23], {...}) -> [{"name": "jane", "age": 23}]`

`eval(family[].name.proper(@), {...}) -> ["Frank", "Jane"]`

`eval(family[].age | avg(@), {...}) -> 22.5`

## 10.9. Filter Expressions

```
bracketExpression
: '[' '*' ']'
| '[' slice ']'
| '[' ']'
| '[' '?' expression ']'
| '[' expression ']'
;
```

A filter expression is defined by a **bracketExpression** where the bracket contents are prefixed with a question mark character (?). A filter expression provides a way to select JSON elements based on a comparison to another expression. A filter expression is evaluated as follows:

- For each element in an array evaluate the **expression** against the element.
- If the expression evaluates to a truth-like value, the item (in its entirety) is added to the result array.
- If the expression does not evaluate to a truth-like value it is excluded from the result array.

A filter expression is defined only for a JSON array. Attempting to evaluate a filter expression against any other type will return **null**.

## Examples

```
eval(
  foo[?a < b],
  {
    "foo": [
      {"a": "char", "b": "bar"},
      {"a": 2, "b": 1},
      {"a": 1, "b": 2},
      {"a": false, "b": "1"},
      {"a": 10, "b": "12"}
    ]
  })
->
[ {"a": 1, "b": 2},
  {"a": false, "b": "1"},
  {"a": 10, "b": "12"} ]
```

The five elements in the foo array are evaluated against **a < b**:

- The first element resolves to the comparison **"char" < "bar"**, and because these types are string, the comparison of code points returns **false**, and the first element is excluded from the result array.
- The second element resolves to **2 < 1**, which is **false**, so the second element is excluded from the result array.
- The third expression resolves to **1 < 2** which evaluates to **true**, so the third element is included in the result array.
- The fourth expression resolves to **false < "1"**. Since the left hand operand is boolean, both operands are coerced to numbers and evaluated as: **0 < 1** and so the fourth element included in the result array.
- The final expression resolves to **10 < "12"**. Since we have mixed operands, the operands are coerced to numbers and evaluated as: **10 < 12** and the last element is included in the result array.

## Examples

```
eval(foo[?bar==10], {"foo": [{"bar": 1}, {"bar": 10}]}) -> [{"bar": 10}]
eval([?bar==10], [{"bar": 1}, {"bar": 10}]) -> [{"bar": 10}]
eval(foo[?a==b], {"foo": [{"a": 1, "b": 2}, {"a": 2, "b": 2}]})
-> [{"a": 2, "b": 2}]
```

## 10.10. Pipe Expressions

```
pipeExpression = expression '|' expression
```

A pipe expression combines two expressions, separated by the pipe (|) character. It is similar to a [chained expression](#) with two distinctions:

1. Any expression can be used on the right hand side. A chained expression restricts the type of expression that can be used on the right hand side.
2. A pipe expression **stops projections on the left hand side from propagating to the right hand side**. If the left expression creates a projection, the right hand side will receive the array underlying the projection.

For example, given the following data

```
{"foo": [{"bar": ["first1", "second1"]}, {"bar": ["first2", "second2"]}]}
```

The expression `foo[*].bar` gives the result of

```
[
  [
    "first1",
    "second1"
  ],
  [
    "first2",
    "second2"
  ]
]
```

The first part of the expression, `foo[*]`, creates a projection. At this point, the remaining expression, `bar` is projected onto each element of the array created from `foo[*]`. If you project the `[0]` expression, you will get the first element from each sub array. The expression `foo[*].bar[0]` will return

```
["first1", "first2"]
```

If you instead wanted **only** the first sub array, `["first1", "second1"]`, you can use a pipe expression.

```
foo[*].bar[0] -> ["first1", "first2"]
foo[*].bar | [0] -> ["first1", "second1"]
```

## Examples

```
eval(foo | bar, {"foo": {"bar": "baz"}}) -> "baz"
eval(foo[*].bar | [0], {
  "foo": [{"bar": ["first1", "second1"]},
          {"bar": ["first2", "second2"]}]) -> ["first1", "second1"]
eval(foo | [0], {"foo": [0, 1, 2]}) -> 0
```

# 11. Functions

```
functionExpression
: NAME '(' functionArg (',' functionArg)* ')'
| NAME '(' ')'
;

functionArg
: expression
| expressionType
;

expressionType : '&' expression ;
```

json-formula has a robust set of built-in [functions](#). Each function has a signature that defines the expected types of the input and the type of the returned output.

```
return_type function_name(type $argname)
return_type function_name2(type1|type2 argname)
```

Functions support the set of standard json-formula [data types](#). If the resolved arguments cannot be coerced to match the types specified in the signature, an **invalid-type** error occurs.

As a shorthand, the type **any** is used to indicate that the function argument can be of any of (**array|object|number|string|boolean|null**).

The expression type, (denoted by **&expression**), is used to specify an expression that is not immediately evaluated. Instead, a reference to that expression is provided to the function being called. The function can then apply the expression reference as needed. It is semantically similar to an [anonymous function](#). See the [sortBy\(\)](#) function for an example of the expression type.

The result of the `functionExpression` is the result returned by the function call. If a `functionExpression` is evaluated for a function that does not exist, an `unknown-function` error is raised.

Functions can either have a specific arity or be variadic with a minimum number of arguments. If a `functionExpression` is encountered where the arity does not match, or the minimum number of arguments for a variadic function is not provided, or too many arguments are provided, then an `invalid-arity` error is raised.

Functions are evaluated in applicative order. Each argument must be an expression, each argument expression must be evaluated before evaluating the function. The function is then called with the evaluated function arguments. The one exception to this rule is the `if(expr, result1, result2)` function. In this case either the `result1` expression or the `result2` expression is evaluated, depending on the outcome of `expr`.

Consider this example using the `abs()` function. Given:

```
{"foo": -1, "bar": "2"}
```

Evaluating `abs(foo)` works as follows:

1. Evaluate the input argument against the current data:

```
eval(foo, {"foo": -1, "bar": "2"}) -> -1
```

2. Coerce the type of the resolved argument if needed. In this case `-1` is of type `number` so no coercion is needed.
3. Validate the type of the coerced argument. In this case `-1` is of type `number` so it passes the type check.
4. Call the function with the resolved argument:

```
abs(-1) -> 1
```

Below is the same steps for evaluating `abs(bar)`:

1. Evaluate the input argument against the current data:

```
eval(bar, {"foo": -1, "bar": "2"}) -> "2"
```

2. Attempt to coerce the result to the required number type. In this case, coerce `"2"` to `2`.



3. Validate the type of the coerced argument. In this case `2` is of type `number` so it passes the type check.
4. Call the function with the resolved and coerced argument:

```
abs(2) -> 2
```

Function expressions are allowed as the child element of a [chained expression](#). The function is then evaluated in the context of the parent expression result. For example: `[1,2,3].sum(@) -> 6`.

When the parent expression is a [projection](#) a chained function expression will be applied to every element in the projection. For example, given the input data of `["1", "2", "3", "notanumber", true]`, the following expression can be used to convert all elements to numbers:

```
eval([].toNumber(@), ["1", "2", "3", "notanumber", null, true]) ->
[1,2,3,0,null,1]
```

## 11.1. Function Reference

### 11.1.1. abs

**abs(value) ⇒ number**

#### Description

Find the absolute (non-negative) value of the provided argument `value`.

#### Returns

number - If `value < 0`, returns `-value`, otherwise returns `value`

#### Parameters

Param	Type	Description
value	number	a numeric value

#### Example

```
abs(-1) // returns 1
```

### 11.1.2. acos

**acos(cosine) ⇒ number**

#### Description

Compute the inverse cosine (in radians) of a number.

#### Returns

number - The inverse cosine angle in radians between 0 and PI

#### Parameters

Param	Type	Description
cosine	number	A number between -1 and 1, inclusive, representing the angle's cosine value.

#### Example

```
acos(0) => 1.5707963267948966
```

### 11.1.3. and

**and(firstOperand, [...additionalOperands]) ⇒ boolean**

#### Description

Finds the logical AND result of all parameters. If the parameters are not boolean they will be [cast to boolean](#). Note the related <>.

#### Returns

boolean - The logical result of applying AND to all parameters

#### Parameters

Param	Type	Description
firstOperand	any	logical expression
[...additionalOperands]	any	any number of additional expressions

#### Example

```
and(10 > 8, length("foo") < 5) // returns true
and(`null`, length("foo") < 5) // returns false
```

### 11.1.4. asin

**asin(sine) ⇒ number**

#### Description

Compute the inverse sine (in radians) of a number.

#### Returns

number - The inverse sine angle in radians between  $-\pi/2$  and  $\pi/2$

#### Parameters

Param	Type	Description
sine	number	A number between -1 and 1, inclusive, representing the angle's sine value.

#### Example

```
Math.asin(0) => 0
```

### 11.1.5. atan2

**atan2(y, x) ⇒ number**

#### Description

Compute the angle in the plane (in radians) between the positive x-axis and the ray from (0, 0) to the point (x, y)

#### Returns

number - The angle in radians (between  $-\pi$  and  $\pi$ ), between the positive x-axis and the ray from (0, 0) to the point (x, y).

#### Parameters

Param	Type	Description
y	number	The y coordinate of the point

Param	Type	Description
x	number	The x coordinate of the point

### Example

```
atan2(20,10) => 1.1071487177940904
```

## 11.1.6. avg

**avg(elements) ⇒ number**

### Description

Finds the average of the elements in an array. An empty array will return an average of **null**.

### Returns

number - average value

### Parameters

Param	Type	Description
elements	Array.<number>	array of numeric values

### Example

```
avg([]) // returns null
avg([1, 2, 3]) // returns 2
```

## 11.1.7. casefold

**casefold(input) ⇒ string**

### Description

Generates a lower-case string of the **input** string using locale-specific mappings. e.g. Strings with German letter ß (eszett) can be compared to "ss"

### Returns

string - A new string converted to lower case

### Parameters

Param	Type	Description
input	string	string to casefold

### Example

```
casefold("AbC") // returns "abc"
```

## 11.1.8. ceil

**ceil(num) ⇒ integer**

### Description

Finds the next highest integer value of the argument **num** by rounding up if necessary.

### Returns

integer - The smallest integer greater than or equal to num

### Parameters

Param	Type	Description
num	number	numeric value

### Example

```
ceil(10) // returns 10
ceil(10.4) // return 11
```

## 11.1.9. codePoint

**codePoint(str) ⇒ integer**

### Description

Retrieve the code point from the first character of a string

### Returns

integer - unicode code point value

### Parameters

Param	Type	Description
str	string	source string

## Example

```
codePoint("ABC") // 65
```

### 11.1.10. contains

**contains(subject, search) ⇒ boolean**

#### Description

Determines if the given **subject** contains the **search** string. If **subject** is an array, this function returns true if one of the elements in the array is equal to the provided **search** value. If **subject** is a string, this function returns true if the string contains the **search** value.

#### Returns

boolean - true if found

#### Parameters

Param	Type	Description
subject	array   string	the subject in which the element has to be searched
search	string   boolean   number   date   null	element to find. If <b>subject</b> is an array, search for an exact match for <b>search</b> in the array. If <b>subject</b> is a string, <b>search</b> will be coerced to a string.

## Example

```
contains([1, 2, 3, 4], 2) // returns true
contains([1, 2, 3, 4], -1) // returns false
contains("Abcd", "d") // returns true
contains("Abcd", "x") // returns false
```

### 11.1.11. cos

**cos(angle) ⇒ number**

#### Description

Compute the cosine (in radians) of a number.

## Returns

number - The cosine of the angle, between -1 and 1, inclusive.

## Parameters

Param	Type	Description
angle	number	A number representing an angle in radians

## Example

```
cos(1.0471975512) => 0.4999999999970535
```

## 11.1.12. datedif

**datedif(start\_date, end\_date, unit) ⇒ integer**

### Description

Return difference between two date values. The measurement of the difference is determined by the **unit** parameter. One of:

- **y** the number of whole years between start\_date and end\_date
- **m** the number of whole months between start\_date and end\_date.
- **d** the number of days between start\_date and end\_date
- **md** the number of days between start\_date and end\_date after subtracting whole months.
- **ym** the number of whole months between start\_date and end\_date after subtracting whole years.
- **yd** the number of days between start\_date and end\_date, assuming start\_date and end\_date were no more than one year apart

## Returns

integer - The number of days/months/years difference

## Parameters

Param	Type	Description
start_date	number	The starting date. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

Param	Type	Description
end_date	number	The end date – must be greater or equal to start_date.
unit	string	

## Example

```
datedif(datetime(2001, 1, 1), datetime(2003, 1, 1), "y") // returns 2
datedif(datetime(2001, 6, 1), datetime(2003, 8, 15), "D") // returns 805
// 805 days between June 1, 2001, and August 15, 2003
datedif(datetime(2001, 6, 1), datetime(2003, 8, 15), "YD") // returns 75
// 75 days between June 1 and August 15, ignoring the years of the dates (75)
```

## 11.1.13. datetime

**datetime(year, month, day, [hours], [minutes], [seconds], [milliseconds]) ⇒ number**

### Description

Generate a date/time value from individual date/time parts. If any of the units are greater than their normal range, the overflow will be added to the next greater unit. e.g. specifying 25 hours will increment the day value by 1.

### Returns

number - A date/time numeric value to be used with other date/time functions

### Parameters

Param	Type	Default	Description
year	integer		The year to use for date construction. Values from 0 to 99 map to the years 1900 to 1999. All other values are the actual year
month	integer		The month: beginning with 1 for January to 12 for December.
day	integer		The day of the month.



Param	Type	Default	Description
[hours]	integer	0	Integer value between 0 and 23 representing the hour of the day.
[minutes]	integer	0	Integer value representing the minute segment of a time.
[seconds]	integer	0	Integer value representing the second segment of a time.
[milliseconds]	integer	0	Integer value representing the millisecond segment of a time.

### Example

```
datetime(2010, 10, 10) // returns representation of October 10, 2010
datetime(2010, 2, 28) // returns representation of February 28, 2010
datetime(2023,13,5) | year(@) & "/" & month(@) // returns 2024/1
```

## 11.1.14. day

**day(date) ⇒ integer**

### Description

Finds the day of the month for a date value

### Returns

integer - The day of the month ranging from 1 to 31.

### Parameters

Param	Type	Description
date	number	date/time generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

## Example

```
day(datetime(2008,5,23)) // returns 23
```

### 11.1.15. deepScan

**deepScan(object, name) ⇒ Array.<any>**

#### Description

Searches a nested hierarchy of objects to return an array of key values that match a **name**. The name can be either a key into an object or an array index. This is similar to the Descendant Accessor operator (**. .**) from [E4X](#).

#### Returns

Array.<any> - The array of matched elements

#### Parameters

Param	Type	Description
object	object   array	The starting object or array where we start the search
name	string   integer	The name (or index position) of the elements to find

## Example

```
deepScan({a : {b1 : {c : 2}, b2 : {c : 3}}}, "c") // returns [2, 3]
```

### 11.1.16. endsWith

**endsWith(subject, suffix) ⇒ boolean**

#### Description

Determines if the **subject** string ends with a specific **suffix**

#### Returns

boolean - true if the **suffix** value is at the end of the **subject**

#### Parameters

Param	Type	Description
subject	string	source string in which to search
suffix	string	search string

### Example

```
endsWith("Abcd", "d") // returns true
endsWith("Abcd", "A") // returns false
```

## 11.1.17. entries

**entries(obj) ⇒ Array.<any>**

### Description

Returns an array of **[key, value]** pairs from an object. The **fromEntries()** function may be used to convert the array back to an object.

### Returns

Array.<any> - an array of arrays where each child array has two elements representing the key and value of a pair

### Parameters

Param	Type	Description
obj	object	source object

### Example

```
entries({a: 1, b: 2}) // returns [["a", 1], ["b", 2]]
```

## 11.1.18. eomonth

**eomonth(startDate, monthAdd) ⇒ integer**

### Description

Finds the serial number of the end of a month, given **startDate** plus **monthAdd** months

### Returns

integer - the number of days in the computed month

## Parameters

Param	Type	Description
startDate	number	The base date to start from. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.
monthAdd	integer	Number of months to add to start date

## Example

```
ecomonth(datetime(2011, 1, 1), 1) | [month(@), day(@)] // returns [2, 28]
ecomonth(datetime(2011, 1, 1), -3) | [month(@), day(@)] // returns [10, 31]
```

## 11.1.19. exp

**exp(x) ⇒ number**

### Description

Finds e (the base of natural logarithms) raised to a power. (i.e.  $e^x$ )

### Returns

number - e (the base of natural logarithms) raised to power x

## Parameters

Param	Type	Description
x	number	A numeric expression representing the power of e.

## Example

```
exp(10) // returns 22026.465794806718
```

## 11.1.20. false

**false() ⇒ boolean**

## Description

Return constant boolean false value. Expressions may also use the JSON literal: ``false``

## Returns

boolean - constant boolean value `false`

## 11.1.21. find

**find(query, text, [start]) ⇒ integer | null**

## Description

finds and returns the index of query in text from a start position

## Returns

integer | null - The position of the found string, null if not found.

## Parameters

Param	Type	Default	Description
query	string		string to search
text	string		text to be searched
[start]	integer	0	zero-based position to start searching

## Example

```
find("m", "abm") // returns 2
find("M", "abMcdM", 3) // returns 5
find("M", "ab") // returns `null`
find("M", "abMcdM", 2) // returns 2
```

## 11.1.22. floor

**floor(num) ⇒ integer**

## Description

Calculates the next lowest integer value of the argument `num` by rounding down if necessary.

## Returns

integer - The largest integer smaller than or equal to num

## Parameters

Param	Type	Description
num	number	numeric value

## Example

```
floor(10.4) // returns 10  
floor(10) // returns 10
```

### 11.1.23. fromCodePoint

**fromCodePoint(codePoint) ⇒ number**

#### Description

Create a string created from a code point.

#### Returns

number - A string from a given code point

## Parameters

Param	Type	Description
codePoint	integer	An integer between 0 and 0x10FFFF (inclusive) representing a Unicode code point.

## Example

```
fromCodePoint(65) // "A"  
fromCodePoint(65) == "\u0041" // true
```

### 11.1.24. fromEntries

**fromEntries(pairs) ⇒ object**

#### Description

Returns an object by transforming a list of key-value **pairs** into an object. **fromEntries()** is the inverse operation of **entries()**.



## Returns

integer - value between 0 and 23

## Parameters

Param	Type	Description
date	number	The datetime/time for which the hour is to be returned. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

## Example

```
hour(datetime(2008,5,23,12, 0, 0)) // returns 12
hour(time(12, 0, 0)) // returns 12
```

## 11.1.27. if

**if(condition, result1, result2) ⇒ any**

### Description

Return one of two values **result1** or **result2**, depending on the **condition**

## Returns

any - either result1 or result2

## Parameters

Param	Type	Description
condition	any	boolean result of a logical expression
result1	any	if condition is true
result2	any	if condition is false

## Example

```
if(true(), 1, 2) // returns 1
if(false(), 1, 2) // returns 2
```



### 11.1.28. join

**join(glue, stringsarray) ⇒ string**

#### Description

Combines all the elements from the provided array, joined together using the **glue** argument as a separator between each.

#### Returns

string - String representation of the array

#### Parameters

Param	Type	Description
glue	string	
stringsarray	Array.<string>	array of strings or values that can be coerced to strings

#### Example

```
join(",", ["a", "b", "c"]) // returns "a,b,c"  
join(" and ", ["apples", "bananas"]) // returns "apples and bananas"
```

### 11.1.29. keys

**keys(obj) ⇒ array**

#### Description

Generates an array of the keys of the input object. If the object is null, the value return an empty array

#### Returns

array - the array of all the key names

#### Parameters

Param	Type	Description
obj	object	the object to examine

#### Example

```
keys({a : 3, b : 4}) // returns ["a", "b"]
```

### 11.1.30. left

**left(subject, [elements])** ⇒ string | array

#### Description

Return a substring from the start of a string or the left-most elements of an array

#### Parameters

Param	Type	Default	Description
subject	string   array		The source text/array of characters/elements
[elements]	integer	1	number of elements to pick

#### Example

```
left("Sale Price", 4) // returns "Sale"  
left("Sweden") // returns "S"  
left([4, 5, 6], 2) // returns [4, 5]
```

### 11.1.31. length

**length(subject)** ⇒ integer

#### Description

Calculates the length of the input argument based on types:

- string: returns the number of code points
- array: returns the number of array elements
- object: returns the number of key-value pairs

#### Returns

integer - the length of the input subject

#### Parameters

Param	Type	Description
subject	string   array   object	subject whose length to calculate

### Example

```
length('[]') // returns 0
length('') // returns 0
length("abcd") // returns 4
length([1, 2, 3, 4]) // returns 4
length({}) // returns 0
length({a : 3, b : 4}) // returns 2
```

## 11.1.32. log

**log(num) ⇒ number**

### Description

Compute the natural logarithm (base e) of a number

### Returns

number - The natural log value

### Parameters

Param	Type	Description
num	number	A number greater than zero

### Example

```
log(10) // 2.302585092994046
```

## 11.1.33. log10

**log10(num) ⇒ number**

### Description

Compute the base 10 logarithm of a number.

### Returns

number - The base 10 log result

## Parameters

Param	Type	Description
num	number	A number greater than or equal to zero

## Example

```
log10(100000) // 5
```

### 11.1.34. lower

**lower(input) ⇒ string**

#### Description

Converts all the alphabetic characters in a string to lowercase.

#### Returns

string - the lower case value of the input string

## Parameters

Param	Type	Description
input	string	input string

## Example

```
lower("E. E. Cummings") // returns e. e. cummings
```

### 11.1.35. map

**map(expr, elements) ⇒ array**

#### Description

Apply an expression to every element in an array and return the array of results. An input array of length N will return an array of length N.

#### Returns

array - the mapped array

## Parameters

Param	Type	Description
expr	expression	expression to evaluate
elements	array	array of elements to process

## Example

```
map(&(@ + 1), [1, 2, 3, 4]) // returns [2, 3, 4, 5]
map(&length(@), ["doe", "nick", "chris"]) // returns [3, 4, 5]
```

## 11.1.36. max

**max(collection) ⇒ number**

### Description

Calculates the largest value in the provided **collection** arguments. If all collections are empty **null** is returned. `max()` can work on numbers or strings. If a mix of numbers and strings are provided, all values will be coerced to the type of the first value.

### Returns

number - the largest value found

## Parameters

Param	Type	Description
collection	Array.<number>   Array.<string>	array in which the maximum element is to be calculated

## Example

```
max([1, 2, 3], [4, 5, 6], 7) // returns 7
max(`[]`) // returns null
max(["a", "a1", "b"]) // returns "b"
```

## 11.1.37. merge

**merge(...args) ⇒ object**

### Description

Accepts one or more objects, and returns a single object with all objects merged. The first object is

copied, and then each key value pair from each subsequent object are added to the first object. Duplicate keys in subsequent objects will override those found in earlier objects.

### Returns

object - The combined object

### Parameters

Param	Type
...args	object

### Example

```
merge({a: 1, b: 2}, {c : 3, d: 4}) // returns {a :1, b: 2, c: 3, d: 4}
merge({a: 1, b: 2}, {a : 3, d: 4}) // returns {a :3, b: 2, d: 4}
```

## 11.1.38. mid

**mid(subject, startPos, length) ⇒ string | array**

### Description

Extracts a substring from source text, or a subset of an array. or in case of array, extracts a subset of the array from start till the length number of elements. Returns null if the **startPos** is greater than the length of the array

### Returns

string | array - The resulting substring or array subset

### Parameters

Param	Type	Description
subject	string   array	the text string or array of characters or elements to extract.
startPos	integer	the zero-position of the first character or element to extract.

Param	Type	Description
length	integer	The number of characters or elements to return from the string or array. If greater than the length of <b>subject</b> the argument is set to the length of the subject.

### Example

```
mid("Fluid Flow", 0, 5) // returns "Fluid"
mid("Fluid Flow", 6, 20) // returns "Flow"
mid("Fluid Flow", 20, 5) // returns ""
mid([0,1,2,3,4,5,6,7,8,9], 2, 3) // returns [2,3,4]
```

## 11.1.39. min

**min(collection) ⇒ number**

### Description

Calculates the smallest value in the input arguments. If all collections are empty **null** is returned. min() can work on numbers or strings. If a mix of numbers and strings are provided, the type of the first value will be used.

### Parameters

Param	Type	Description
collection	...Array.<number>   Array.<string>   number   string	to search for the minimum value

### Example

```
min([1, 2, 3], [4, 5, 6], 7) // returns 1
min(`[]`) // returns null
min(["a", "a1", "b"]) // returns "a"
```

## 11.1.40. minute

**minute(date) ⇒ integer**

## Description

Extract the minute (0 through 59) from a time/datetime representation

## Returns

integer - Number of minutes in the time portion of the date/time value

## Parameters

Param	Type	Description
date	number	A datetime/time value. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

## Example

```
minute(datetime(2008,5,23,12, 10, 0)) // returns 10  
minute(time(12, 10, 0)) // returns 10
```

## 11.1.41. mod

**mod(dividend, divisor) ⇒ number**

## Description

Return the remainder when one number is divided by another number.

## Returns

number - Computes the remainder of **dividend/divisor**. If **dividend** is negative, the result will also be negative.

## Parameters

Param	Type	Description
dividend	number	The number for which to find the remainder.
divisor	number	The number by which to divide number.

## Example

```
mod(3, 2) // returns 1
```



```
mod(-3, 2) // returns -1
```

## 11.1.42. month

**month(date) ⇒ number**

### Description

Finds the month of a date.

### Returns

number - The month number as an integer, ranging from 1 (January) to 12 (December).

### Parameters

Param	Type	Description
date	number	source date value. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

### Example

```
month(datetime(2008,5,23)) // returns 5
```

## 11.1.43. not

**not(value) ⇒ boolean**

### Description

Compute logical NOT of a value. If the parameter is not boolean it will be [cast to boolean](#) Note the related [unary NOT operator](#).

### Returns

boolean - The logical NOT applied to the input parameter

### Parameters

Param	Type	Description
value	any	any data type

### Example

```
not(length("bar") > 0) // returns false
not(false()) // returns true
not("abcd") // returns false
not("") // returns true
```

### 11.1.44. notNull

**notNull(...argument) ⇒ any**

#### Description

Finds the first argument that does not resolve to `null`. This function accepts one or more arguments, and will evaluate them in order until a non-null argument is encountered. If all arguments values resolve to null, then return a null value.

#### Parameters

Param	Type
...argument	any

#### Example

```
notNull(1, 2, 3, 4, `null`) // returns 1
notNull(`null`, 2, 3, 4, `null`) // returns 2
```

### 11.1.45. now

**now() ⇒ number**

#### Description

Retrieve the current date/time.

#### Returns

number - representation of current date/time as a number

### 11.1.46. null

**null() ⇒ boolean**

#### Description

Return constant null value. Expressions may also use the JSON literal: ``null``

## Returns

boolean - True

### 11.1.47. or

**or(first, [...operand]) ⇒ boolean**

## Description

Determines the logical OR result of a set of parameters. If the parameters are not boolean they will be [cast to boolean](#). Note the related `<>`.

## Returns

boolean - The logical result of applying OR to all parameters

## Parameters

Param	Type	Description
first	any	logical expression
[...operand]	any	any number of additional expressions

## Example

```
or((x / 2) == y, (y * 2) == x) // true
```

### 11.1.48. power

**power(a, x) ⇒ number**

## Description

Computes **a** raised to a power **x**. ( $a^x$ )

## Parameters

Param	Type	Description
a	number	The base number – can be any real number.
x	number	The exponent to which the base number is raised.

## Example

```
power(10, 2) // returns 100 (10 raised to power 2)
```

### 11.1.49. proper

**proper(text) ⇒ string**

#### Description

Apply proper casing to a string. Proper casing is where the first letter of each word is converted to an uppercase letter and the rest of the letters in the word converted to lowercase.

#### Returns

string - source string with proper casing applied.

#### Parameters

Param	Type	Description
text	string	source string

#### Example

```
proper("this is a TITLE") // returns "This Is A Title"  
proper("2-way street") // returns "2-Way Street"  
proper("76BudGet") // returns "76Budget"
```

### 11.1.50. random

**random() ⇒ number**

#### Description

Generate a pseudo random number.

#### Returns

number - A value greater than or equal to zero, and less than one.

#### Example

```
random() // 0.022585461160693265
```

### 11.1.51. reduce

**reduce(expr, elements, initialValue) ⇒ any**

## Description

Executes a user-supplied reducer expression on each element of an array, in order, passing in the return value from the expression from the preceding element. The final result of running the reducer across all elements of the input array is a single value. The expression can access the following properties of the current object:

- **accumulated**: accumulated value based on the previous expression. For the first array element use the **initialValue** parameter. If not provided, then **null**
- **current**: current element to process
- **index**: index of the current element in the array
- **array**: original array

## Parameters

Param	Type	Description
expr	expression	reducer expression to be executed on each element
elements	array	array of elements on which the expression will be evaluated
initialValue	any	the accumulated value to pass to the first array element

## Example

```
reduce(&(accumulated + current), [1, 2, 3]) // returns 6
// find maximum entry by age
reduce(
  &max(@.accumulated.age, @.current.age),
  [{age: 10, name: "Joe"}, {age: 20, name: "John"}], @[0].age
)
reduce(&accumulated * current, [3, 3, 3], 1) // returns 27
```

## 11.1.52. register

**register(functionName, expr) ⇒ Object**

## Description

Register a function. The registered function may take one parameter. If more parameters are needed, combine them in an array or object.

## Returns

Object - returns an empty object

## Parameters

Param	Type	Description
functionName	string	Name of the function to register
expr	expression	Expression to execute with this function call

## Example

```
register("product", &@[0] * @[1]) // can now call: product([2,21]) => returns 42
```

## 11.1.53. replace

**replace(text, start, length, replacement) ⇒ string**

### Description

Generates text where an old text is substituted at a given start position and length, with a new text.

### Parameters

Param	Type	Description
text	string	original text
start	integer	zero-based index in the original text from where to begin the replacement.
length	integer	number of characters to be replaced
replacement	string	string to insert at the start index

## Example

```
replace("abcdefghijk", 5, 5, "*") // returns abcde*k  
replace("2009",2,2,"10") // returns 2010  
replace("123456",0,3,"@") // returns @456
```

### 11.1.54. rept

**rept(text, count) ⇒ string**

#### Description

Return text repeated **count** times.

#### Returns

string - Text generated from the repeated text

#### Parameters

Param	Type	Description
text	string	text to repeat
count	integer	number of times to repeat the text

#### Example

```
rept("x", 5) // returns "xxxxx"
```

### 11.1.55. reverse

**reverse(argument) ⇒ array**

#### Description

Reverses the order of an array or string

#### Returns

array - The resulting reversed array or string

#### Parameters

Param	Type	Description
argument	string   array	the source to be reversed

#### Example

```
reverse(["a", "b", "c"]) // returns ["c", "b", "a"]
```

## 11.1.56. right

**right(subject, [elements]) ⇒ string | array**

### Description

Generates a string from the right-most characters of a string or a subset of elements from the end of an array

### Returns

string | array - The extracted characters or array subset Returns null if the number of elements is less than 0

### Parameters

Param	Type	Default	Description
subject	string   array		The text/array containing the characters/elements to extract
[elements]	integer	1	number of elements to pick

### Example

```
right("Sale Price", 4) // returns "rice"
right("Sweden") // returns "n"
right([4, 5, 6], 2) // returns [5, 6]
```

## 11.1.57. round

**round(num, precision) ⇒ number**

### Description

Round a number to a specified precision:

- If **precision** is greater than zero, round to the specified number of decimal places.
- If **precision** is 0, round to the nearest integer.
- If **precision** is less than 0, round to the left of the decimal point.

### Returns

number - rounded value



## Parameters

Param	Type	Description
num	number	number to round
precision	integer	precision to use for the rounding operation.

## Example

```
round(2.15, 1) // returns 2.2
round(626.3,-3) // returns 1000 (Rounds 626.3 to the nearest multiple of 1000)
round(626.3, 0) // returns 626
round(1.98,-1) // returns 0 (Rounds 1.98 to the nearest multiple of 10)
round(-50.55,-2) // -100 (round -50.55 to the nearest multiple of 100)
```

## 11.1.58. search

**search(findText, withinText, [startPos]) ⇒ array**

### Description

Perform a wildcard search. The search is case-sensitive and supports two forms of wildcards: **\*** finds a sequence of characters and **?** finds a single character. To use **\*** or **?** as text values, precede them with a tilde (~) character. Note that the wildcard search is not greedy. e.g. **search("a\*b", "abb")** will return **[0, "ab"]** Not **[0, "abb"]**

### Returns

array - returns an array with two values:

- The start position of the found text and the text string that was found.
- If a match was not found, an empty array is returned.

## Parameters

Param	Type	Default	Description
findText	string		the search string – which may include wild cards.
withinText	string		The string to search.
[startPos]	integer	0	The zero-based position of withinText to start searching.

## Example

```
search("a?c", "acabc") // returns [2, "abc"]
```

## 11.1.59. second

**second(date) ⇒ integer**

### Description

Extract the seconds of the time value in a time/datetime representation

### Returns

integer - The number of seconds: 0 through 59

### Parameters

Param	Type	Description
date	number	datetime/time for which the second is to be returned. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

## Example

```
second(datetime(2008,5,23,12, 10, 53)) // returns 53  
second(time(12, 10, 53)) // returns 53
```

## 11.1.60. sign

**sign(num) ⇒ number**

### Description

Computes the sign of a number passed as argument.

### Returns

number - returns 1 or -1, indicating the sign of **num**. If the **num** is 0, it will be returned 0.

### Parameters

Param	Type	Description
num	number	any number

### Example

```
sign(5) // 1
sign(-5) // -1
sign(0) // 0
```

## 11.1.61. sin

**sin(angle) ⇒ number**

### Description

Computes the sine of a number in radians

### Returns

number - The sine of **angle**, between -1 and 1, inclusive

### Parameters

Param	Type	Description
angle	number	A number representing an angle in radians.

### Example

```
sin(0) // 0
sin(1) // 0.8414709848078965
```

## 11.1.62. sort

**sort(list) ⇒ Array.<number> | Array.<string>**

### Description

This function accepts an array of strings or numbers and returns an array with the elements in sorted order. String sorting is based on code points. Sort is not locale-sensitive.

### Returns

Array.<number> | Array.<string> - The ordered result

## Parameters

Param	Type	Description
list	Array.<number>   Array.<string>	to be sorted

## Example

```
sort([1, 2, 4, 3, 1]) // returns [1, 1, 2, 3, 4]
```

### 11.1.63. sortBy

**sortBy(elements, expr) ⇒ array**

#### Description

Sort an array using an expression to find the sort key. For each element in the array, the expression is applied and the resulting value is used as the sort value. If the result of evaluating the expression against the current array element results in type other than a number or a string, a type-error will occur.

#### Returns

array - The sorted array

## Parameters

Param	Type	Description
elements	array	Array to be sorted
expr	expression	The comparison expression

## Example

```
// returns ["e", "def", "abcd"]
sortBy(["abcd", "e", "def"], &length(@))

// returns [{year: 1910}, {year: 2010}, {year: 2020}]
sortBy([{year: 2010}, {year: 2020}, {year: 1910}], &year)

// returns [5, -10, -11, -15, 30]
sortBy([-15, 30, -10, -11, 5], &abs(@))
```

### 11.1.64. split

**split(string, separator) ⇒ Array.<string>**

#### Description

Split a string into an array, given a separator

#### Returns

Array.<string> - The array of separated strings

#### Parameters

Param	Type	Description
string	string	string to split
separator	string	separator where the split(s) should occur

#### Example

```
split("abcdef", "") // returns ["a", "b", "c", "d", "e", "f"]
split("abcdef", "e") // returns ["abcd", "f"]
```

### 11.1.65. sqrt

**sqrt(num) ⇒ number**

#### Description

Find the square root of a number

#### Returns

number - The calculated square root value

#### Parameters

Param	Type	Description
num	number	source number

#### Example

```
sqrt(4) // returns 2
```

### 11.1.66. startsWith

**startsWith(subject, prefix) ⇒ boolean**

#### Description

Determine if a string starts with a prefix.

#### Returns

boolean - true if **prefix** matches the start of **subject**

#### Parameters

Param	Type	Description
subject	string	string to search
prefix	string	prefix to search for

#### Example

```
startsWith("jack is at home", "jack") // returns true
```

### 11.1.67. stdev

**stdev(numbers) ⇒ number**

#### Description

Estimates standard deviation based on a sample. **stdev** assumes that its arguments are a sample of the entire population. If your data represents a entire population, then compute the standard deviation using [stdevp](#).

#### Returns

number - [Standard deviation](#)

#### Parameters

Param	Type	Description
numbers	Array.<number>	The array of numbers comprising the population

#### Example

```
stdev([1345, 1301, 1368]) // returns 34.044089061098404
```

```
stdevp([1345, 1301, 1368]) // returns 27.797
```

### 11.1.68. stdevp

**stdevp(numbers) ⇒ number**

#### Description

Calculates standard deviation based on the entire population given as arguments. **stdevp** assumes that its arguments are the entire population. If your data represents a sample of the population, then compute the standard deviation using **stdev**.

#### Returns

number - Calculated standard deviation

#### Parameters

Param	Type	Description
numbers	Array.<number>	The array of numbers comprising the population

#### Example

```
stdevp([1345, 1301, 1368]) // returns 27.797
stdev([1345, 1301, 1368]) // returns 34.044
```

### 11.1.69. substitute

**substitute(text, old, new, [which]) ⇒ string**

#### Description

Generates a string from the input **text**, with text **old** replaced by text **new** (when searching from the left). If there is no match, or if **old** has length 0, **text** is returned unchanged. Note that **old** and **new** may have different lengths. If **which** < 1, return **text** unchanged

#### Returns

string - replaced string

#### Parameters

Param	Type	Description
text	string	The text for which to substitute characters.
old	string	The text to replace.
new	string	The text to replace <b>old</b> with.
[which]	integer	The one-based occurrence of <b>old</b> text to replace with <b>new</b> text. If <b>which</b> parameter is omitted, every occurrence of <b>old</b> is replaced with <b>new</b> .

### Example

```
substitute("Sales Data", "Sales", "Cost") // returns "Cost Data"
substitute("Quarter 1, 2008", "1", "2", 1) // returns "Quarter 2, 2008"
substitute("Quarter 1, 1008", "1", "2", 2) // returns "Quarter 1, 2008"
```

## 11.1.70. sum

**sum(collection) ⇒ number**

### Description

Calculates the sum of the provided array. An empty array will produce a return value of 0.

### Returns

number - The computed sum

### Parameters

Param	Type	Description
collection	Array.<number>	array of numbers

### Example

```
sum([1, 2, 3]) // returns 6
```

## 11.1.71. tan

**tan(angle) ⇒ number**



## Description

Computes the tangent of a number in radians

## Returns

number - The tangent of **angle**

## Parameters

Param	Type	Description
angle	number	A number representing an angle in radians.

## Example

```
tan(0) // 0
tan(1) // 1.5574077246549023
```

## 11.1.72. time

**time(hours, [minutes], [seconds])** ⇒ number

## Description

Construct and returns a time value.

## Returns

number - Returns a date/time value representing the fraction of the day consumed by the given time

## Parameters

Param	Type	Default	Description
hours	integer		Zero-based integer value between 0 and 23 representing the hour of the day.
[minutes]	integer	0	Zero-based integer value representing the minute segment of a time.

Param	Type	Default	Description
[seconds]	integer	0	Zero-based integer value representing the seconds segment of a time.

### Example

```
time(12, 0, 0) | [hour(@), minute(@), second(@)] // returns [12, 0, 0]
```

## 11.1.73. toArray

**toArray(arg) ⇒ array**

### Description

Converts the provided argument to an array. The conversion happens as per the following rules:

- array - Returns the provided value.
- number/string/object/boolean/null - Returns a one element array containing the argument.

### Returns

array - The resulting array

### Parameters

Param	Type	Description
arg	any	parameter to turn into an array

### Example

```
toArray(1) // returns [1]
toArray(null()) // returns [null]
```

## 11.1.74. today

**today() ⇒ number**

### Description

Returns a date/time value representing the start of the current day. i.e. midnight

## Returns

number - today at midnight

## 11.1.75. toNumber

**toNumber(arg, [base])** ⇒ number

## Description

Converts the provided arg to a number as per the [type coercion rules](#).

## Parameters

Param	Type	Default	Description
arg	string   number   boolean   null		to convert to number
[base]	integer	10	The base to use. One of: 2, 8, 10, 16. Defaults to 10.

## Example

```
toNumber(1) // returns 1
toNumber("10") // returns 10
toNumber({a: 1}) // fails
toNumber(true()) // returns 1
toNumber("10f") // returns 0
toNumber("FF", 16) // returns 255
```

## 11.1.76. toString

**toString(arg)** ⇒ string

## Description

Converts the provided argument to a string as per the [type coercion rules](#).

## Returns

string - The result string

## Parameters

Param	Type	Description
arg	any	Value to be converted to a string

### Example

```
toString(1) // returns "1"
toString(true()) // returns "true"
toString({sum: 12 + 13}) // "{\"sum\":25}"
```

## 11.1.77. trim

**trim(text) ⇒ string**

### Description

Remove leading and trailing spaces, and replace all internal multiple spaces with a single space.

### Returns

string - trimmed string

### Parameters

Param	Type	Description
text	string	string to trim

### Example

```
trim("  ab  c  ") // returns "ab c"
```

## 11.1.78. true

**true() ⇒ boolean**

### Description

Return constant boolean true value. Expressions may also use the JSON literal: ``true``

### Returns

boolean - True

## 11.1.79. trunc

**trunc(numA, [numB])** ⇒ number

### Description

Truncates a number to an integer by removing the fractional part of the number.

### Returns

number - Truncated value

### Parameters

Param	Type	Default	Description
numA	number		number to truncate
[numB]	integer	0	A number specifying the number of decimal digits to preserve.

### Example

```
trunc(8.9) // returns 8
trunc(-8.9) // returns -8
trunc(8.912, 2) // returns 8.91
```

## 11.1.80. type

**type(subject)** ⇒ string

### Description

Finds the type name of the given **subject** argument as a string value. The return value will be one of the following:

- number
- string
- boolean
- array
- object
- null

## Returns

string - The type name

## Parameters

Param	Type	Description
subject	any	type to evaluate

## Example

```
type(1) // returns "number"  
type("") // returns "string"
```

## 11.1.81. unique

**unique(input) ⇒ array**

### Description

Find the set of unique elements within an array

## Returns

array - array with duplicate elements removed

## Parameters

Param	Type	Description
input	array	input array

## Example

```
unique([1, 2, 3, 4, 1, 1, 2]) // returns [1, 2, 3, 4]
```

## 11.1.82. upper

**upper(input) ⇒ string**

### Description

Converts all the alphabetic characters in a string to uppercase.

## Returns

string - the upper case value of the input string

## Parameters

Param	Type	Description
input	string	input string

## Example

```
upper("abcd") // returns "ABCD"
```

## 11.1.83. value

**value(object, index) ⇒ any**

### Description

Perform an indexed lookup on an object or array

### Returns

any - the result of the lookup – or **null** if not found.

## Parameters

Param	Type	Description
object	object   array	on which to perform the lookup
index	string   integer	a named child for an object or an integer offset for an array

## Example

```
value({a: 1, b:2, c:3}, "a") // returns 1  
value([1, 2, 3, 4], 2) // returns 3
```

## 11.1.84. values

**values(obj) ⇒ array**

### Description

Generates an array of the values of the provided object. Note that because JSON objects are inherently unordered, the values associated with the provided object are also unordered.

### Returns

array - array of the key values

## Parameters

Param	Type	Description
obj	object	source object

## Example

```
values({a : 3, b : 4}) // returns [3, 4]
```

## 11.1.85. weekday

**weekday(date, [returnType])** ⇒ integer

### Description

Extract the day of the week from a date. The specific numbering of the day of week is controlled by the **returnType** parameter:

- 1 : Sunday (1), Monday (2), ..., Saturday (7)
- 2 : Monday (1), Tuesday (2), ..., Sunday(7)
- 3 : Monday (0), Tuesday (2), ..., Sunday(6)

### Returns

integer - day of the week

## Parameters

Param	Type	Default	Description
date	number		datetime for which the day of the week is to be returned. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.
[returnType]	integer	1	Determines the representation of the result

## Example

```
weekday(datetime(2006,5,21)) // 1
```



```
weekday(datetime(2006,5,21), 2) // 7
weekday(datetime(2006,5,21), 3) // 6
```

### 11.1.86. year

**year(date) ⇒ integer**

#### Description

Finds the year of a datetime value

#### Returns

integer - The year value

#### Parameters

Param	Type	Description
date	number	input date/time value. Date/time values can be generated using the <a href="#">datetime</a> , <a href="#">today</a> , <a href="#">now</a> and <a href="#">time</a> functions.

#### Example

```
year(datetime(2008,5,23)) // returns 2008
```

### 11.1.87. zip

**zip(...arrays) ⇒ array**

#### Description

Generates a convolved (zipped) array containing grouped arrays of values from the array arguments from index 0, 1, 2, etc. This function accepts a variable number of arguments. The length of the returned array is equal to the length of the shortest array.

#### Returns

array - An array of arrays with elements zipped together

#### Parameters

Param	Type	Description
...arrays	array	array of arrays to zip together

## Example

```
zip([1, 2, 3], [4, 5, 6, 7]) // returns [[1, 4], [2, 5], [3, 6]]
```

# 12. Integrations

The json-formula API allows integrations to customize various json-formula behaviors.

## 12.1. Globals

By default, json-formula has one global symbol: `@`. A host may inject additional global identifiers. These identifiers must be prefixed with the dollar (`$`) symbol.

## Examples

Given: a global symbol:

```
{
  "$days": [
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
    "Sunday"
  ]
}
```

```
eval(value($days, weekday(datetime(date.year, date.month, date.day), 3)),
{
  "date": {
    "year": 2023,
    "month": 9,
    "day": 13
  }
}) -> "Wednesday"
```

## 12.2. Specify locale

The default locale for json-formula is `en-US`. A host may specify an alternate locale. Overall, the locale setting has little effect on processing. One specific area that is affected is the behavior of the `casefold()` function.

## 12.3. Custom toNumber

In various contexts, json-formula converts values to numbers. A host may provide its own `toNumber()` function that json-formula will use in place of the default functionality. For example, a custom `toNumber()` could make use of locale-specific date formats to attempt to convert a string to a date value, or could allow currency values e.g., "\$123.45" to be converted to number.

## 12.4. Additional Functions

A host may provide its own set of functions to augment the base set provided by json-formula.

## 12.5. Hidden Properties

A host system may construct its source JSON data with complex properties that have nested structure that can be found through explicit navigation, but will not be found through normal tree searching. Here is an example of how this can be configured in JavaScript:

```
function createField(id, value) {
  class Field {
    valueOf() { return value; }

    toString() { return value.toString(); }

    toJSON() { return value; }
  }
  const f = new Field();
  Object.defineProperty(f, '$id', { get: () => id });
  Object.defineProperty(f, '$value', { get: () => value });

  return f;
}

const json = {
  "street": createField("abc123", "Maple Street"),
  "city": createField("def456", "New York")
}
```

Given this configuration, these search results are possible:

```
street -> "Maple Street"
street.$value -> "Maple Street"
street.$id -> "abc123"
type(street) -> "string"
keys(street) -> []
```