json-formula Specification

Table of Contents

Abstract

This document is the specification for json-formula.

1. Notation

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

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

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

```
    search(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. Structured data in, structured data out.

2. Data Types

json-formula supports the same types supported by JSON:

- number (integers and double-precision floating-point format in JSON)
- 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 (denoted by `&expression`)

Implementations can map the corresponding JSON types to their language equivalent. For example, a JSON `null` could map to `None` in python, and `nil` in ruby and go.

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 flatten 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 error shall be emitted — most often an `invalid-type` error.

Examples

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

2.2. Type Coercion Rules

Provided Type	Expected Type	Result
number	string	number converted to a string. Similar to JavaScript `toString()`

Provided Type	Expected Type	Result
boolean	string	"true" or "false"
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. Allow for locale-specific currency symbols to be ignored.
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	An object where the keys are the array index positions and the values are the array values.
null	object	Empty object
number	boolean	zero is false. All other numbers are true

Provided Type	Expected Type	Result
string	boolean	Empty string is false, populated strings are true
array	boolean	Empty array is false, populated arrays are true
object	boolean	Empty object is false, populated objects are true
null	boolean	false

Examples

```
search("\"$123.00\" + 1", {}) -> 124.00"
search("truth is " & `true`, {}) -> "truth is true"
search(2 + `true`, {}) -> 3
search(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

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

4. Floating Point Precision

json-formula implementations are expected to use Double-precision floating-point format to represent numbers. 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 → 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. Grammar

The grammar is specified using Antlr.

```
grammar JsonFormula;

formula : expression EOF ;

expression
  : expression '.' chainedExpression
  | expression indexExpression
  | indexExpression
  | expression ('*' | '/' | '&' | '~') expression
  | expression ('+' | '-') expression
  | expression COMPARATOR expression
  | expression '&&' expression
  | expression '||' expression
  | identifier
```

```
  | '!' expression
  | '-' expression
  | '(' expression ')'
  | wildcard
  | multiSelectArray
  | multiSelectObject
  | JSON_FRAGMENT
  | functionExpression
  | expression '|' expression
  | STRING
  | (REAL_OR_EXPONENT_NUMBER | INT)
  | currentNode
  ;

chainedExpression
  : identifier
  | multiSelectArray
  | multiSelectObject
  | functionExpression
  | wildcard
  ;

wildcard : '*' ;

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

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

keyvalExpr : identifier ':' expression ;

indexExpression
  : '[' '*' ']'
  | '[' 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 | [bfnrt\\`'"/]);

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'
  | [1-9] [0-9]*
  ;

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

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

```
  ;
```

In addition to the grammar, there is the following token precedence that goes 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: `-`

6. 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-Za-z_$`.

An identifier can also be quoted. This is necessary when an identifier has characters not specified in the `NAME` grammar rule. In this situation, an identifier is specified with a single quote, followed by any number of characters, followed by a single quote. Any valid string can be used between single quoted, include JSON supported escape sequences, and six character unicode escape sequences.

Note that any identifier that does not start with `A-Za-z_$` **must** be quoted.

Examples

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

7. 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.

8. SubExpressions

```
subExpression: expression '.' chainedExpression

chainedExpression
  : identifier
  | multiSelectArray
  | multiSelectObject
  | functionExpression
  | wildcard
  ;

wildcard : '*' ;
```

A subexpression is a combination of two expressions separated by the `.` char. A subexpression is evaluated as follows:

- Evaluate the expression on the left with the original JSON document.
- Evaluate the expression on the right with the result of the left expression evaluation.

In pseudo-code

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

A subexpression 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 = search(foo, {"foo": {"bar": "baz"}}) -> {"bar": "baz"}
result = search(bar, {"bar": "baz"}) -> "baz"
```

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

Additional examples

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

9. Bracket Expressions

```
expression: expression indexExpression

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

From the `indexExpression` construction, an index expression is where the brackets contents provide access to the elements in an array or object.

When brackets are used with an object, the bracket contents are expected to be an expression that resolves to the name of a property. In the simple case, `foo["bar"]` is equivalent to `foo.bar`.

Most commonly, brackets are used with arrays. With arrays, bracketed expressions are expected to return an integer value to return an index in the array. Indexing is 0 based. The index of 0 refers to the first element of the array. A negative number is a valid index. A negative number 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. If an index expression refers to an index that is greater than the length of the array, a value of `null` is returned.

Using a '*' character within a `indexExpression` is discussed below in the Wildcard Expressions section.

Using an expression prefixed with a "?" character within a `indexExpression` is discussed below in the Filter Expressions section.

9.1. 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]`, but each component is optional and can be omitted.

> 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 how many indices to skip after each element is selected from the array. An array of 1 (the default step) will not skip any indices. A step value of 2 will skip every other index while extracting elements from an array. A step value of -1 will extract values in reverse order from the array.

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 it 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

```
search(foo["bar"], {"foo": {"bar": 21}}) -> 21
search([0:4:1], [0, 1, 2, 3]) -> [0, 1, 2, 3]
search([0:4], [0, 1, 2, 3]) -> [0, 1, 2, 3]
search([0:3], [0, 1, 2, 3]) -> [0, 1, 2]
search([:2], [0, 1, 2, 3]) -> [0, 1]
search([::2], [0, 1, 2, 3]) -> [0, 2]
search([::-1], [0, 1, 2, 3]) -> [3, 2, 1, 0]
search([-2:], [0, 1, 2, 3]) -> [2, 3]
```

9.2. 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 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 now the new current result.

Once the flattening operation has been performed, subsequent operations are projected onto the flattened array with the same semantics as a wildcard expression. Thus the difference between `[*]` and `[]` is that `[]` will first flatten sub-arrays in the current result.

Examples

```
search([0], ["first", "second", "third"]) -> "first"
search([-1], ["first", "second", "third"]) -> "third"
search([100], ["first", "second", "third"]) -> null
search(foo[0], {"foo": ["first", "second", "third"]}) -> "first"
search(foo[100], {"foo": ["first", "second", "third"]}) -> null
search(foo[0][0], {"foo": [[0, 1], [1, 2]]}) -> 0
search(foo[], {"foo": [[0, 1], [1, 2]]}) -> [0,1,1,2]
```

10. Operators

10.1. 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.
- 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, follow the type coercion rules.

The comparison semantics for each operator are defined in the discussion in type coercion.

10.1.1. Equality Operators

Two representations of the equality and inequality operators are supported: `=` and `==` are equivalent in functionality. Both variations are supported provide familiarity to users with experience with similar grammars. Similarly, `!=` and `<>` function identically.

For `string/number/true/false/null` types, equality is an exact match. A `string` is equal to another `string` if they they have the exact same sequence of code points. 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 (given two JSON objects `x` and `y`, 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 (given two arrays `x` and `y`, for each `i` from `0` until `length(x)`, `x[i] == y[i]`).

10.1.2. Ordering Operators

- 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, follow the type coercion rules.

10.2. Numeric operators

The following operators operate on numbers:

- addition: `+`
- subtraction: `-`
- multiplication `*`
- division: `/`

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

10.3. Concatenation Operator

The concatenation operator takes two string operands and combines them to form a single string.

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

10.4. Union Operator

The union operator (~) combines the contents of two arrays into a single array.

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

10.5. Array Operators

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

- When these operators are provided arrays for both operands, the operator is applied to 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, the operator is applied with the scalar against each element in the array.

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

11. Or Expressions

```
orExpression = expression '||' expression
```

An or expression will evaluate to either the left expression or the right expression. If the evaluation of the left expression can be coerced to a true value, it is used as the return value. If the left expression cannot be coerced to a true value, then the evaluation of the right expression 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

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

12. And Expressions

```
andExpression = expression '&&' expression
```

An and expression will evaluate to either the left expression or the right expression. If the expression on the left hand side is a truth-like value, then the value on the right hand side is returned. Otherwise the result of the expression on the left hand side 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

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

13. Paren Expressions

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

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

Examples

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

14. Not Expressions

```
  notExpression = '!' expression
```

A `notExpression` negates the result of an expression. If the expression results in a truth-like value, a `notExpression` will change this value to `false`. If the expression results in a false-like value, a `notExpression` will change this value to `true`.

Examples

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

15. MultiSelect Array

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

A multiselect expression is used to extract a subset of elements from a JSON object or array. There are two version of multiselect, one in which the multiselect expression is enclosed in `{…}` and one which is enclosed in `[…]`. This section describes the `[…]` version. Within the start and closing characters is one or more expressions separated by a comma. Each expression will be evaluated against the JSON document. Each returned element will be the result of evaluating the expression. A `multiSelectArray` with `N` expressions will result in an array of length `N`. Given a multiselect expression `[expr-1,expr-2,…,expr-n]`, the evaluated expression will return `[evaluate(expr-1), evaluate(expr-2), …, evaluate(expr-n)]`.

Examples

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

16. MultiSelect Object

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

A `multiSelectObject` expression is similar to a `multiSelectArray` expression, except that an object is created instead of an array. A `multiSelectObject` expression also 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 `multiSelectObject` will correspond to a single key value pair in the created object. Unlike the `multiSelectArray`, a `multiSelectObject` may be empty.

Examples

Given a `multiSelectObject` 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:

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

17. Wildcard Expressions

There are two forms of wildcard expression:

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

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

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

```
expression : expression '.' chainedExpression

chainedExpression
  : identifier
  | multiSelectArray
  | multiSelectObject
  | functionExpression
  | wildcard
  ;

wildcard : '*' ;
```

A wildcard expression is an expression of either `.*` or `[*]`. A wildcard expression can return multiple elements, and the remaining expressions are evaluated against each returned element from a wildcard expression. The `[*]` syntax applies to an array type and the `.*` syntax applies to an object type.

The `[*]` syntax (referred to as an array wildcard expression) will return all the elements in an array. Any subsequent expressions will be evaluated against each individual element. Given an expression `[*].childExpr`, and an array of N elements, the evaluation of this expression would be `[childExpr(el-0), childExpr(el-2), …, childExpr(el-N)]`. This is referred to as a **projection**, and the `childExpr` expression is projected onto the elements of the resulting array.

Once a projection has been created, all subsequent expressions are projected onto the resulting array.

The `*` syntax (referred to as an object wildcard expression) will return an array of the object element's values. Any subsequent expression will be evaluated against each individual element in the array (this is also referred to as a **projection**).

An array wildcard expression is valid only for the JSON array type. If an array wildcard expression is applied to any other JSON type, a value of `null` is returned.

Similarly, an object wildcard expression is valid only for the JSON object type. 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. Implementations are not required to return the object values in any specific order.

Examples

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

18. JSON Literal Expressions

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

A JSON literal expression is an expression that allows arbitrary JSON objects to be specified. This is useful in filter expressions as well as multi select objects (to create arbitrary key value pairs), but is allowed anywhere an expression is allowed. The specification includes JSON literals. Implementations should use an existing JSON parser to parse these literals. Note that the \ 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

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

19. String Literals

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

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

fragment UNICODE
  : 'u' HEX HEX HEX HEX
  ;

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

A `STRING` is an expression that allows for a literal string value to be specified. A literal string supports the same character escape sequences as strings in JSON. e.g. a unicode character 'A' could be specified as `\u0041`.

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

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

20. Number literals

```
numberLiteral = REAL_OR_EXPONENT_NUMBER | INT

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

INT
  : '0'
  | [1-9] [0-9]*
  ;

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

As with literal strings, json-formula allows literal numbers in expressions. Number literals follow the same syntax rules as numeric values in JSON with the exception that number literals may omit a leading zero. For example, `.123` is not valid JSON, but is allowed aas a number literal. Note that 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.

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

Examples

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

21. 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`.

21.1. 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}
  ]
}

search(@.family[0].name, {...}) => "frank"

search(family[].[left(@.name), age], {...}) =>
   [["f", 22], ["j", 23]]

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

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

search(family[].name.proper(@), {...}) => ["Frank", "Jane"]

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

22. Filter Expressions

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

A filter expression is defined by a `indexExpression` 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. Otherwise 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

```
search(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 right hand operand is coerced to the same type as the left hand operand (numeric) and evaluated as: `10 < 12` and the last element is included in the result array.

Examples

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

23. Function Expressions

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

functionArg
  : expression
  | expressionType
  ;

expressionType : '&' expression ;
```

Functions allow users to easily transform and filter data in json-formula expressions.

24. Function Evaluation

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 result1 or result2 is evaluated, depending on the outcome of `expr`. 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, the json-formula implementation must indicate to the caller that an `unknown-function` error occurred. How and when this error is raised is implementation specific, but implementations should indicate to the caller that this specific error occurred.

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, then implementations must indicate to the caller than an `invalid-arity` error occurred. How and when this error is raised is implementation specific.

Each function signature declares the types of its input parameters. If any type constraints are not met, implementations must indicate that an `invalid-type` error occurred.

In order to accommodate type constraints, function implementations will attempt to coerce parameters to the correct type. If implicit coercion is not sufficient, explicit functions are provided to convert values to other types (`toString()`, `toNumber()`).

Function expressions are also allowed as the child element of a sub expression. This allows functions to be used with projections, which can enable functions to be applied to every element in a projection. For example, given the input data of `["1", "2", "3", "notanumber", true]`, the following expression can be used to convert (and filter) all elements to numbers

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

This provides a simple mechanism to explicitly convert types when needed.

25. Built-in Functions

json-formula has a robust set of built-in functions that operate on different data types. 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 do not match the types specified in the signature, an `invalid-type` error occurs.

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

json-formula functions are required to attempt to coerce values to the required type and are required to type check the resulting coerced arguments. Specifying an invalid type for a function argument will result in a `invalid-type` error.

The expression type, denoted by `&expression`, is used to specify a 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.

Similar to how arrays can specify a type within an array using the `type[]` syntax, expressions can specify their resolved type using `expression→type` syntax. This means that the resolved type of the function argument must be an expression that itself will resolve to `type`.

To demonstrate the above points, 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:

```
search(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
```

5. The value of `1` is the resolved value of the function expression `abs(foo)`

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

1. Evaluate the input argument against the current data:

```
search(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
```

5. The value of `2` is the resolved value of the function expression `abs(bar)`

26. Pipe Expressions

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

A pipe expression combines two expressions, separated by the `|` character. It is similar to a `chainedExpression` with two important distinctions:

1. Any expression can be used on the right hand side. A `chainedExpression` restricts the type of expression that can be used on the right hand side.

2. A `pipeExpression` **stops projections on the left hand side for propagating to the right hand side**. If the left expression creates a projection, it does **not** apply to the right hand side.

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 `pipeExpression`

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

Examples

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

27. Integrations

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

27.1. 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"
    ]
  }
```

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

27.2. 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.

27.3. Custom toNumber

In various contexts, json-formula converts values to numbers. The default string-to-number functionality will make a modest attempt to convert currencies or date values to number—however this functionality is rudimentary and is not necessarily consistent in different locales. 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.

27.4. Additional Functions

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

27.5. 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. In JavaScript, this can be configured as:

```
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) => []
```

28. Function Reference

28.1. abs

abs(num) ⇒ number

Find the absolute value of the provided argument `value`.

Returns: number - the absolute value of the `value` argument

Param	Type	Description
num	number	a numeric value

Example

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

28.2. and

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

Finds the logical AND result of all parameters. If the parameters are not boolean they will be cast to boolean as per the type coercion rules

Returns: boolean - The logical result of applying AND to all 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
```

28.3. avg

avg(elements) ⇒ number

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

Returns: number - average value

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

Example

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

28.4. casefold

casefold(input) ⇒ string

Generates a lower-case string of the `input` string using locale-specific mappings. e.g. Strings with German letter ß can be compared to ``ss''

Returns: string - A new string converted to lower case

Param	Type	Description
input	string	string to casefold

Example

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

28.5. charCode

charCode(codePoint) ⇒ number

Create a string created from the specified code unit.

Returns: number - A string from a given code point

Param	Type	Description
codePoint	integer	unicode code point value

Example

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

28.6. ceil

ceil(num) ⇒ integer

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

Param	Type	Description
num	number	numeric value

Example

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

28.7. codePoint

codePoint(str) ⇒ integer

Retrieve the code point from the first character of a string

Returns: integer - unicode code point value

Param	Type	Description
str	string	source string

Example

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

28.8. contains

contains(subject, search) ⇒ boolean

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

Param	Type	Description
subject	array | string	the subject in which the element has to be searched
search	string | boolean | number | date	element to search

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
```

28.9. datedif

datedif(start_date, end_date, unit) ⇒ integer

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

Param	Type	Description
start_date	number	The starting date. Date/time values can be generated using the datetime, today, now and time functions.
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)
```

28.10. datetime

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

Return a date/time value.

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

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
```

28.11. day

day(date) ⇒ integer

Finds the day of a date

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

Param	Type	Description
date	number	of the day you are trying to find. Date/time values can be generated using the datetime, today, now and time functions.

Example

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

28.12. deepScan

deepScan(object, name) ⇒ Array.<any>

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 JSONPath deep scan operator (..)

Returns: Array.<any> - The array of matched elements

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]
```

28.13. endsWith

endsWith(subject, suffix) ⇒ boolean

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

Returns: boolean - true if the `suffix` value is at the end of the `subject`

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
```

28.14. entries

entries(obj) ⇒ Array.<any>

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

Param	Type	Description
obj	object	source object

Example

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

28.15. eomonth

eomonth(startDate, monthAdd) ⇒ integer

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

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

Example

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

28.16. exp

exp(x) ⇒ number

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

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

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

Example

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

28.17. false

false() ⇒ boolean

Return constant boolean false value. Note that expressions may also use the JSON literal false: `\'false\'`

Returns: boolean - constant boolean value `false`

28.18. find

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

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.

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
```

28.19. fromEntries

fromEntries(pairs) ⇒ object

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

Returns: object - An object constructed from the provided key-value pairs

Param	Type	Description
pairs	Array.<any>	A nested array of key-value pairs to create the object from

Example

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

28.20. floor

floor(num) ⇒ integer

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

Param	Type	Description
num	number	numeric value

Example

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

28.21. hour

hour(date) ⇒ integer

Extract the hour from a date/time representation

Returns: integer - value between 0 and 23

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

Example

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

28.22. if

if(condition, result1, result2) ⇒ any

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

Returns: any - either result1 or result2

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
```

28.23. join

join(glue, stringsarray) ⇒ string

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

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"
```

28.24. keys

keys(obj) ⇒ array

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

Param	Type	Description
obj	object	the object to examine

Example

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

28.25. left

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

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

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]
```

28.26. length

length(subject) ⇒ integer

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

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
```

28.27. lower

lower(input) ⇒ string

Converts all the alphabetic characters in a string to lowercase. If the value is not a string it will be converted into string.

Returns: string - the lower case value of the input string

Param	Type	Description
input	string	input string

Example

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

28.28. map

map(expr, elements) ⇒ array

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

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]
```

28.29. max

max(collection) ⇒ number

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 with be coerced to the type of the first value.

Returns: number - the largest value found

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"
```

28.30. merge

merge(...args) ⇒ object

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

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}
```

28.31. mid

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

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

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.
length	integer	The number of characters or elements to return from the string or array. If greater then the length of `subject` 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]
```

28.32. min

min(collection) ⇒ number

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.

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"
```

28.33. minute

minute(date) ⇒ integer

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

Param	Type	Description
date	number	A datetime/time value. Date/time values can be generated using the datetime, today, now and time functions.

Example

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

28.34. mod

mod(dividend, divisor) ⇒ number

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.

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
```

28.35. month

month(date) ⇒ number

Finds the month of a date.

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

Param	Type	Description
date	number	source date value. Date/time values can be generated using the datetime, today, now and time functions.

Example

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

28.36. not

not(value) ⇒ boolean

Compute logical NOT of a value. If the parameter is not boolean it will be cast to boolean as per the type coercion rules. Note the related unary not operator: !

Returns: boolean - The logical NOT applied to the input parameter

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
```

28.37. notNull

notNull(...argument) ⇒ any

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.

Param	Type
…argument	any

Example

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

28.38. now

now() ⇒ number

Retrieve the current date/time.

Returns: number - representation of current date/time as a number

28.39. null

null() ⇒ boolean

Return constant null value. Note that expressions may also use the JSON literal null: `\'null\'`

Returns: boolean - True

28.40. or

or(first, [...operand]) ⇒ boolean

Determines the logical OR result of a set of parameters. If the parameters are not boolean they will be cast to boolean as per the type coercion rules. Note the related `or' operator: `A || B`.

Returns: boolean - The logical result of applying OR to all parameters

Param	Type	Description
first	any	logical expression

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

Example

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

28.41. power

power(a, x) ⇒ number

Computes a raised to a power x. (ax)

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)
```

28.42. proper

proper(text) ⇒ string

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.

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"
```

28.43. random

random() ⇒ number

Generate a pseudo random number.

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

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

28.44. reduce

reduce(expr, elements, initialValue) ⇒ any

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

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
```

28.45. register

register(functionName, expr) ⇒ Object

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

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
```

28.46. replace

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

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

Param	Type	Description
text	string	original text
start	integer	zero-based index in the original text from where to begin the replacement.

Param	Type	Description
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
```

28.47. rept

rept(text, count) ⇒ string

Return text repeated `count` times.

Returns: string - Text generated from the repeated text

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

Example

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

28.48. reverse

reverse(argument) ⇒ array

Reverses the order of an array or string

Returns: array - The resulting reversed array or string

Param	Type	Description
argument	string | array	the source to be reversed

Example

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

28.49. right

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

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

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]
```

28.50. round

round(num, precision) ⇒ number

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

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)
```

28.51. search

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

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.

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"]
```

28.52. second

second(date) ⇒ integer

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

Returns: integer - The number of seconds: 0 through 59

Param	Type	Description
date	number	datetime/time for which the second is to be returned. Date/time values can be generated using the datetime, today, now and time functions.

Example

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

28.53. sort

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

This function accepts an array of strings or numbers and returns a re-orderd array with the elements in sorted order. String sorting is based on code points. Locale is not taken into account.

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

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

Example

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

28.54. sortBy

sortBy(elements, expr) ⇒ array

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

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

Example

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

// returns [{year: 1910}, {year: 2010}, {year: 2020}]
sortBy([{year: 2010}, {year: 2020}, {year: 1910}], &year)
sortBy([-15, 30, -10, -11, 5], &abs(@)) // [5, -10, -11, -15, 30]
```

28.55. split

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

split a string into an array, given a separator

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"]
```

28.56. sqrt

sqrt(num) ⇒ number

Return the square root of a number

Returns: number - the calculated square root value

Param	Type	Description
num	number	source number

Example

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

28.57. startsWith

startsWith(subject, prefix) ⇒ boolean

Determine if a string starts with a prefix.

Returns: boolean - true if `prefix` matches the start of `subject`

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

Example

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

28.58. stdev

stdev(numbers) ⇒ number

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

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
```

28.59. stdevp

stdevp(numbers) ⇒ number

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

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
```

28.60. substitute

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

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

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

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"
```

28.61. sum

sum(collection) ⇒ number

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

Returns: number - The sum of elements

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

Example

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

28.62. time

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

Construct and returns a time value.

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

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.
[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]
```

28.63. toArray

toArray(arg) ⇒ array

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

Param	Type	Description
arg	any	parameter to turn into an array

Example

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

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

28.64. today

today() ⇒ number

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

Returns: number - today at midnight

28.65. toNumber

toNumber(arg) ⇒ number

Converts the provided arg to a number. The conversion happens as per the type coercion rules.

Param	Type	Description
arg	any	to convert to number

Example

```
toNumber(1) // returns 1
toNumber("10") // returns 10
toNumber({a: 1}) // returns null
toNumber(true()) // returns 1
toNumber("10f") // returns 0
```

28.66. toString

toString(arg) ⇒ string

Converts the provided argument to a string. The conversion happens as per the type coercion rules.

Returns: string - The result string

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}"
```

28.67. trim

trim(text) ⇒ string

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

Returns: string - trimmed string

Param	Type	Description
text	string	string to trim

Example

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

28.68. true

true() ⇒ boolean

Return constant boolean true value. Note that expressions may also use the JSON literal true: `\'true\'`

Returns: boolean - True

28.69. trunc

trunc(numA, [numB]) ⇒ number

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

Returns: number - Truncated value

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
```

28.70. type

type(subject) ⇒ string

Finds the JavaScript type of the given `subject` argument as a string value. The return value MUST be one of the following:

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

Returns: string - The type of the subject

Param	Type	Description
subject	any	type to evaluate

Example

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

28.71. unique

unique(input) ⇒ array

Find the set of unique elements within an array

Returns: array - array with duplicate elements removed

Param	Type	Description
input	array	input array

Example

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

28.72. upper

upper(input) ⇒ string

Converts all the alphabetic characters in a string to uppercase. If the value is not a string it will be converted into string according to the type coercion rules.

Returns: string - the upper case value of the input string

Param	Type	Description
input	string	input string

Example

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

28.73. value

value(object, index) ⇒ any

Perform an indexed lookup on an object or array

Returns: any - the result of the lookup – or `null` if not found.

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
```

28.74. values

values(obj) ⇒ array

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

Param	Type	Description
obj	object	source object

Example

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

28.75. weekday

weekday(date, [returnType]) ⇒ integer

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

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 datetime, today, now and time 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
```

28.76. year

year(date) ⇒ integer

Finds the year of a datetime value

Returns: integer - The year value

Param	Type	Description
date	number	input date/time value. Date/time values can be generated using the datetime, today, now and time functions.

Example

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

28.77. zip

zip(...arrays) ⇒ array

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

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]]
```