json-formula Specification

John Brinkman

1.0 : 2023-09-18

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.

2.1.1. 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	0
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

2.2.1. Examples

```
search("\"$123.00\" + 1", {}) -> 124.00"
search("truth is " & `true`, {}) -> "truth is true"
search(2 + `true`, {}) -> 3
search(avg("20"), {}) -> 20
search(merge({a: 3}, 4), {}) -> {a:3}
search(merge({a: 3}, ["x", "y"]), {}) -> {"0": "x", "1": "y", "a": 3}
```

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 integer 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
- As implied, 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()`

3.1. 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, as described here.

```
// $antlr-format false
/*
Copyright 2021 Adobe. All rights reserved.
This file is licensed to you under the Apache License, Version 2.0 (the
"License");
you may not use this file except in compliance with the License. You may obtain a
copy
of the License at http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed
under
the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
REPRESENTATIONS
OF ANY KIND, either express or implied. See the License for the specific language
```

```
governing permissions and limitations under the License.
*/

grammar JSONFormula;

formula : expression EOF ;

expression
  : expression '.' chainedExpression # chainExpression
  | expression chainedBracketSpecifier # bracketedExpression
  | indexExpression # bracketExpression
  | expression ('^') expression # powerExpression
  | expression ('*' | '/' | '&' | '~') expression   # multDivExpression
  | expression ('+' | '-') expression   # addSubtractExpression
  | expression COMPARATOR expression # comparisonExpression
  | expression '&&' expression # andExpression
  | expression '||' expression # orExpression
  | identifier # identifierExpression
  | '!' expression # notExpression
  | '-' expression # unaryMinusExpression
  | '(' expression ')' # parenExpression
  | wildcard # wildcardExpression
  | multiSelectList # multiSelectListExpression
  | multiSelectHash # multiSelectHashExpression
  | literal # literalExpression
  | functionExpression # functionCallExpression
  | expression '|' expression # pipeExpression
  | (STRING | RAW_STRING) # rawStringExpression
  | (REAL_OR_EXPONENT_NUMBER | SIGNED_INT) # numberLiteral
  | currentNode # currentNodeExpression
  ;

chainedExpression
  : identifier
  | multiSelectList
  | multiSelectHash
  | functionExpression
  | wildcard
  ;

wildcard : '*' ;

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

multiSelectHash
  : '{' '}' #emptyHash
  | '{' keyvalExpr (',' keyvalExpr)* '}'  #nonEmptyHash
  ;

keyvalExpr : identifier ':' expression ;

indexExpression
  : '[' SIGNED_INT ']' # bracketIndex
  | '[' '*' ']' # bracketStar
  | '[' slice ']' # bracketSlice
```

```
  | '[' ']' # bracketFlatten
  | '[?' expression ']' # select
  ;

chainedBracketSpecifier
  : indexExpression # chainedBracket
  | '[' expression ']' # chainedBracketIndex
  ;

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

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

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

functionArg
  : expression
  | expressionType
  ;

currentNode : '@' ;

expressionType : '&' expression ;

literal : '`' jsonValue '`' ;

identifier
  : NAME
  | QUOTED_NAME
  | JSON_CONSTANT
  ;

JSON_CONSTANT
  : 'true'
  | 'false'
  | 'null'
  ;

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

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

jsonObject
```

```
  : '{' jsonObjectPair (',' jsonObjectPair)* '}'
  | '{' '}'
  ;

jsonObjectPair
  : STRING ':' jsonValue
  ;

jsonArray
  : '[' jsonValue (',' jsonValue)* ']'
  | '[' ']'
  ;

jsonValue
  : STRING # jsonStringValue
  | (REAL_OR_EXPONENT_NUMBER | SIGNED_INT) # jsonNumberValue
  | jsonObject # jsonObjectValue
  | jsonArray # jsonArrayValue
  | JSON_CONSTANT # jsonConstantValue
  ;

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

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

RAW_STRING : '"' (RAW_ESC | ~["\\])* '"' ;

fragment RAW_ESC : '\\' . ;


fragment UNICODE
  : 'u' HEX HEX HEX HEX
  ;

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

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

SIGNED_INT : '-'? INT ;

fragment 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
  | JSON_CONSTANT
  ;

JSON_CONSTANT
  : 'true'
  | 'false'
  | 'null'
  ;

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, than 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 `unquoted-string` grammar rule. In this situation, an identifier is specified with a single quote, followed by any number of `unescaped-char` or `escaped-char` 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.

6.1. 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
  | multiSelectList
  | multiSelectHash
  | 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 sub-expressions: `grandparent.parent.child`.

8.1. 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 chainedBracketSpecifier

chainedBracketSpecifier
  : indexExpression
  | '[' expression ']'
  ;

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

From the `chainedBracketSpecifier` construction, an index expression is the case where the brackets enclose an integer or wildcard (`*`) or expression that returns a value to access elements in a list 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 lists. With lists, bracketed expressions are expected to return an integer, and then follow the semantics of `SIGNED_INT`. When brackets enclose a signed integer (`SIGNED_INT`), indexing is 0 based. The index of 0 refers to the first element of the list. A negative number is a valid index. A negative number indicates that indexing is relative to the end of the list, 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 list. If an index expression refers to an index that is greater than the length of the array, a value of `null` is returned.

For the grammar rule `expression indexExpression` the `expression` is first evaluated, and then return value from the `expression` is given as input to the `indexExpression`.

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.

9.2. 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]
```

9.3. 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 sub-lists in the current result into a single list. The flattening operator has the following semantics:

- Create an empty result list.
- Iterate over the elements of the current result.
- If the current element is not a list, add to the end of the result list.
- If the current element is a list, add each element of the current element to the end of the result list.
- The result list is now the new current result.

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

9.4. 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 on 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`, {}) -> [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 list: `[]`
- Empty object: `{}`
- Empty string: `""`
- False boolean: `false`
- Null value: `null`
- zero value: `0`

11.1. 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 || mylist[-1], {"mylist": ["one", "two"]}) -> "two"
  search(override || mylist[-1], {"mylist": ["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 also 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 [AND](https://en.wikipedia.org/wiki/Truth_table#Logical_conjunction_.28AND.29).

12.1. 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`.

13.1. 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`.

14.1. Examples

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

15. MultiSelect List

```
  multiSelectList : '[' 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 `multi-select-list` with `N` expressions will result in a list 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)]`.

15.1. 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 Hash

```
multi-select-hash = "{" ( keyval-expr ( "," keyval-expr )*)? "}"
keyval-expr       = identifier ":" expression
```

A `multi-select-hash` expression is similar to a `multi-select-list` expression, except that a hash is created instead of a list. A `multi-select-hash` expression also requires key names to be provided, as specified in the `keyval-expr` rule. Given the following rule

```
keyval-expr = 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 `keyval-expr` within the `multi-select-hash` will correspond to a single key value pair in the created hash. Unlike the multi-select-list, a multi-select-hash may be empty.

16.1. Examples

Given a `multi-select-hash` expression `{foo: one.two, bar: bar}` and the data `{"bar": "bar", {"one": {"two": "one-two"}}}`, the expression is evaluated as follows:

1. A hash 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, `[*]` from the `indexExpression` construction:

```
indexExpression
  : '[' SIGNED_INT ']' # bracketIndex
  | '[' '*' ']' # bracketStar
  | '[' slice ']' # bracketSlice
  | '[' ']' # bracketFlatten
  | '[?' expression ']' # select
  ;
```

And `.*` from the chainedExpression construction:

```
expression : expression '.' chainedExpression

chainedExpression
  : identifier
  | multiSelectList
  | multiSelectHash
  | functionExpression
  | wildcard
  ;

wildcard : '*' ;
```

A wildcard expression is a 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 a list type and the .` syntax applies to a hash type.

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

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

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

Note that if any subsequent expression after a wildcard expression returns a `null` value, it is omitted from the final result list.

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

Similarly, a hash wildcard expression is valid only for the JSON object type. If a hash wildcard expression is applied to any other JSON type, a value of `null` is returned. Note that JSON hashes are explicitly defined as unordered. Therefore a hash wildcard expression can return the values associated with the hash in any order. Implementations are not required to return the hash values in any specific order.

17.1. Examples

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

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 hashes (to create arbitrary key value pairs), but is allowed anywhere an expression is allowed. The specification includes the definition for JSON, implementations should use an existing JSON parser to parse literal values. Note that the `\` character must now be escaped in a `json-value` which means implementations need to handle this case

before passing the resulting string to a JSON parser.

18.1. 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. Raw String Literals

```
RAW_STRING : '"' (RAW_ESC | ~["\\])* '"' ;

fragment RAW_ESC : '\\' . ;
```

A raw string is an expression that allows for a literal string value to be specified. The result of evaluating the raw string literal expression is a literal string value. It is a simpler form of a JSON literal expression that is special cased for strings. For example, the following expressions both evaluate to the same value: "foo"

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

As you can see in the examples above, it is meant as a more succinct form of the common scenario of specifying a json literal string value.

In addition, it does not perform any of the additional processing that JSON strings supports including:

- Not expanding unicode escape sequences
- Not expanding newline characters
- Not expanding tab characters or any other escape sequences documented in RFC 4627 section 2.5.

```
  search("foo", {}) -> "foo"
  search(" bar ", {}) -> " bar "
  search("[baz]", {}) -> "[baz]"
  search("[baz]", {}) -> "[baz]"
  search("\u03a6", {}) -> "\u03a6"
  search("\"\\\"", {}) -> "\\"
```

20. Number literals

```
numberLiteral = (REAL_OR_EXPONENT_NUMBER | SIGNED_INT)
REAL_OR_EXPONENT_NUMBER
  : '-'? INT '.' [0-9] + EXP?
  | '-'? INT EXP
  ;

SIGNED_INT : '-'? INT ;

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

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

As with raw strings, json-formula allows literal numbers in expressions. Allowing numbers in expressions does lead to some ambiguity. A bracket with a single signed digit e.g.: `[0]` can be interpreted as a flatten operation or a multi-select list 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]` ``

20.1. 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. Filter Expressions

```
indexExpression
  : '[' SIGNED_INT ']'
  | '[' '*' ']'
  | '[' slice ']'
  | '[' ']'
  | '[?' 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 list. Otherwise it is excluded from the result list. A filter expression is defined only for a JSON array. Attempting to evaluate a filter expression against any other type will return `null`.

21.1. 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 list 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 list.
- The second element resolves to `2 < 1`, which is `false`, so the second element is excluded from the result list.
- The third expression resolves to `1 < 2` which evaluates to `true`, so the third element is included in the result list.

- 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 list.
- 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 list.

21.2. 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}]
```

22. Function Expressions

```
    function-expression = unquoted-string  (
                            no-args  /
                            one-or-more-args )
    no-args             = "(" ")"
    one-or-more-args    = "(" ( function-arg *( "," function-arg ) ) ")"
    function-arg        = expression / current-node / expression-type
    current-node        = "@"
    expression-type     = "&" expression
```

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

22.1. current-node

The `current-node` token can be used to represent the current node being evaluated. The `current-node` token is useful for functions that require the current node being evaluated as an argument. For example, given:

```
{
  "family": [
    {"name": "Joe", "age": 22},
    {"name": "Jane", "age": 23, "occupation": "lawyer"}
  ]
}
```

The following expression creates an array of arrays where each nested array contains the total number

of elements in the object and the 'name' property of the object:

```
    family[].[length(@), name]
```

json-formula assumes that all function arguments operate on the current node unless the argument is a literal. Because of this, an expression such as `@.name` would be equivalent to just `name`.

22.1.1. current-node 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, the value the current node represents MUST change to reflect the node currently being evaluated. When in a projection, the current node value must be changed to the node currently being evaluated by the projection.

23. 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 `function-expression` is the result returned by the function call. If a `function-expression` 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 `function-expression` 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]) ->
[1,2,3,0,1]
```

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

24. Built-in Functions

json-formula has a robust set of built-in functions that operate on different data types. Each function below 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)
```

The list of data types supported by a function are:

- 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
- expression (denoted by `&expression`)

With the exception of the last item, all of the above types correspond to the types provided by JSON.

If the resolved arguments do not match the types specified in the signature, an `invalid-type` error occurs.

The `array` type can further specify requirements on the type of the elements if they want to enforce homogeneous types. The subtype is surrounded by `[type]`, for example, the function signature requires its input argument resolves to an array of numbers

```
    return_type foo(array[number] $argname)
```

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 choose to apply the expression reference as needed. It is semantically similar to an anonymous function. See the `sortBy` function for an example usage of the expression type.

Similarly how arrays can specify a type within a list using the `array[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)`

25. Pipe Expressions

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

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

1. Any expression can be used on the right hand side. A `sub-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 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 list created from `foo[]`. If you project the `[0]` expression, you will get the first element from each sub list. The expression `foo[*].bar[0]` will return

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

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

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

25.1. 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
```

26. Integrations

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

26.1. 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.

26.1.1. 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"
```

26.2. 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.

26.3. 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.

26.4. Additional Functions

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