JMESPath

Release 0.7.1

April 27, 2015

Contents

JSON Matching Expression paths. JMESPath allows you to declaratively specify how to extract elements from a JSON document.

For example, given this document:

```
{"foo": {"bar": "baz"}}
```

The jmespath expression `foo.bar` will return "baz".

Contents:

CHAPTER 1

JMESPath Specification

Warning: This page is deprecated and will be removed in the near future. Go to http://jmespath.org/specification.html for the latest JMESPath specification.

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

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

For simplicity, the jmespath expression and the JSON document are not quoted. For example:

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

In this specification, `null` is used as a return value whenever an expression does not match. `null` is the generic term that maps to the JSON `null` value. Implementations can replace the `null` value with the language equivalent value.

1.1 Grammar

The grammar is specified using ABNF, as described in RFC4234

```
expression        = sub-expression / index-expression / or-expression / identifier
expression        =/ "*" / multi-select-list / multi-select-hash / literal
expression        =/ function-expression / pipe-expression
sub-expression    = expression "." ( identifier /
                                     multi-select-list /
                                     multi-select-hash /
                                     function-expression /
                                     "*" )
or-expression     = expression "||" expression
pipe-expression   = expression "|" expression
index-expression  = expression bracket-specifier / bracket-specifier
multi-select-list = "[" ( expression *( "," expression ) ) "]"
multi-select-hash = "{" ( keyval-expr *( "," keyval-expr ) ) "}"
keyval-expr       = identifier ":" expression
bracket-specifier = "[" (number / "*" / slice-expression) "]" / "[]"
bracket-specifier =/ "[?" list-filter-expr "]"
list-filter-expr  = expression comparator expression
slice-expression  = [number] ":" [number] [ ":" [number] ]
comparator        = "<" / "<=" / "==" / ">=" / ">" / "!="
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

literal           = "`" json-value "`"
literal           =/ "`" 1*(unescaped-literal / escaped-literal) "`"
unescaped-literal = %x20-21 /       ; space !
                        %x23-5A /   ; # - [
                        %x5D-5F /   ; ] ^ _
                        %x61-7A     ; a-z
                        %x7C-10FFFF ; |}~ ...
escaped-literal   = escaped-char / (escape %x60)
number            = ["-"]1*digit
digit             = %x30-39
identifier        = unquoted-string / quoted-string
unquoted-string   = (%x41-5A / %x61-7A / %x5F) *(  ; a-zA-Z_
                        %x30-39  /  ; 0-9
                        %x41-5A /  ; A-Z
                        %x5F    /  ; _
                        %x61-7A)   ; a-z
quoted-string     = quote 1*(unescaped-char / escaped-char) quote
unescaped-char    = %x20-21 / %x23-5B / %x5D-10FFFF
escape            = %x5C   ; Back slash: \
quote             = %x22   ; Double quote: '"'
escaped-char      = escape (
                        %x22 /          ; "    quotation mark  U+0022
                        %x5C /          ; \    reverse solidus U+005C
                        %x2F /          ; /    solidus         U+002F
                        %x62 /          ; b    backspace       U+0008
                        %x66 /          ; f    form feed       U+000C
                        %x6E /          ; n    line feed       U+000A
                        %x72 /          ; r    carriage return U+000D
                        %x74 /          ; t    tab             U+0009
                        %x75 4HEXDIG )  ; uXXXX                U+XXXX

; The ``json-value`` is any valid JSON value with the one exception that the
; ``%x60`` character must be escaped.  While it's encouraged that implementations
; use any existing JSON parser for this grammar rule (after handling the escaped
; literal characters), the grammar rule is shown below for completeness::

json-value = false / null / true / json-object / json-array /
             json-number / json-quoted-string
false = %x66.61.6c.73.65   ; false
null  = %x6e.75.6c.6c      ; null
true  = %x74.72.75.65      ; true
json-quoted-string = %x22 1*(unescaped-literal / escaped-literal) %x22
begin-array     = ws %x5B ws  ; [ left square bracket
begin-object    = ws %x7B ws  ; { left curly bracket
end-array       = ws %x5D ws  ; ] right square bracket
end-object      = ws %x7D ws  ; } right curly bracket
name-separator  = ws %x3A ws  ; : colon
value-separator = ws %x2C ws  ; , comma
ws              = *(%x20 /              ; Space
                    %x09 /              ; Horizontal tab
                    %x0A /              ; Line feed or New line
```

```
                    %x0D                ; Carriage return
                   )
json-object = begin-object [ member *( value-separator member ) ] end-object
member = quoted-string name-separator json-value
json-array = begin-array [ json-value *( value-separator json-value ) ] end-array
json-number = [ minus ] int [ frac ] [ exp ]
decimal-point = %x2E       ; .
digit1-9 = %x31-39         ; 1-9
e = %x65 / %x45            ; e E
exp = e [ minus / plus ] 1*DIGIT
frac = decimal-point 1*DIGIT
int = zero / ( digit1-9 *DIGIT )
minus = %x2D               ; -
plus = %x2B                ; +
zero = %x30                ; 0
```

1.2 Identifiers

```
identifier        = unquoted-string / quoted-string
unquoted-string   = (%x41-5A / %x61-7A / %x5F) *(  ; a-zA-Z_
                        %x30-39  /  ; 0-9
                        %x41-5A /  ; A-Z
                        %x5F    /  ; _
                        %x61-7A)   ; a-z
quoted-string     = quote 1*(unescaped-char / escaped-char) quote
unescaped-char    = %x20-21 / %x23-5B / %x5D-10FFFF
escape            = %x5C   ; Back slash: \
quote             = %x22   ; Double quote: '"'
escaped-char      = escape (
                        %x22 /          ; "    quotation mark  U+0022
                        %x5C /          ; \    reverse solidus U+005C
                        %x2F /          ; /    solidus         U+002F
                        %x62 /          ; b    backspace       U+0008
                        %x66 /          ; f    form feed       U+000C
                        %x6E /          ; n    line feed       U+000A
                        %x72 /          ; r    carriage return U+000D
                        %x74 /          ; t    tab             U+0009
                        %x75 4HEXDIG )  ; uXXXX                U+XXXX
```

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 of 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 double quote, followed by any number of `unescaped-char` or `escaped-char` characters, followed by a double quote. The `quoted-string` rule is the same grammar rule as a JSON string, so any valid string can be used between double 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.

1.2.1 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"
```

1.3 SubExpressions

```
sub-expression    = expression "." ( identifier /
                                     multi-select-list /
                                     multi-select-hash /
                                     function-expression /
                                     "*" )
```

A subexpression is a combination of two expressions separated by the '.' char. A subexpression is evaluted 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 pseudocode:

```
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 subexpressions: `grandparent.parent.child`.

1.3.1 Examples

Given a JSON document: `{"foo": {"bar": "baz"}}`, and a jmespath 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": {"baz": "value"}}) -> null
search(foo.bar.baz, {"foo": {"bar": {"baz": "value"}}}) -> "value"
```

1.4 Index Expressions

```
index-expression  = expression bracket-specifier / bracket-specifier
bracket-specifier = "[" (number / "*" / slice-expression) "]" / "[]"
slice-expression  = [number] ":" [number] [ ":" [number] ]
```

An index expression is used to access elements in a list. 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 bracket-specifier` the `expression` is first evaluated, and then return value from the `expression` is given as input to the `bracket-specifier`.

Using a "*" character within a `bracket-specifier` is discussed below in the `wildcard expressions` section.

1.4.1 Slices

```
slice-expression  = [number] ":" [number] [ ":" [number] ]
```

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.

Note: Slices in JMESPath have the same semantics as python 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 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.

1.4.2 Examples

```
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]
```

1.4.3 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 sublists 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 sublists in the current result.

1.4.4 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
```

1.5 Or Expressions

```
or-expression     = expression "||" expression
```

An or expression will evaluate to either the left expression or the right expression. If the evaluation of the left expression is not false it is used as the return value. If the evaluation of the right expression is not false it is used as the return value. If neither the left or right expression are non-null, then a value of null is returned. A false value corresponds to any of the following conditions:

```
* Empty list: ``[]``
* Empty object: ``{}``
* Empty string: ``""``
* False boolean: ``false``
* Null value: ``null``
```

A true value corresponds to any value that is not false.

1.5.1 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"
```

1.6 MultiSelect List

```
multi-select-list = "[" ( expression *( "," expression ) "]"
```

A multiselect expression is used to extract a subset of elements from a JSON hash. 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 non 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)]`.

1.6.1 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]
```

1.7 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.

1.7.1 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 evaluting 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", "bar": null}
```

1.8 Wildcard Expressions

```
expression        =/ "*"
bracket-specifier = "[" "*" "]"
```

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

1.8.1 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]
```

1.9 Literal Expressions

```
literal           = "`" json-value "`"
literal           =/ "`" 1*(unescaped-literal / escaped-literal) "`"
unescaped-literal = %x20-21 /       ; space !
                    %x23-5A /   ; # - [
                    %x5D-5F /   ; ] ^ _
                    %x61-7A     ; a-z
                    %x7C-10FFFF ; |}~ ...
escaped-literal   = escaped-char / (escape %x60)
```

A 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 ABNF 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.

Note the second literal rule. This is used to specify a string such that double quotes do not have to be included. This means that the literal expression \`"foo"\` is equivalent to \`foo\`.

1.9.1 Examples

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

1.10 Filter Expressions

```
list-filter-expr  = expression comparator expression
comparator        = "<" / "<=" / "==" / ">=" / ">" / "!="
```

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 `list-filter-expr` against the element. If the expression evalutes to `true`, the item (in its entirety) is added to the result list. Otherwise it is excluded from the result list. A filter expression is only defined for a JSON array. Attempting to evaluate a filter expression against any other type will return `null`.

1.10.1 Comparison Operators

The following operations are supported:

- ==, tests for equality.
- !=, tests for inequality.
- <, less than.
- <=, less than or equal to.
- >, greater than.
- >=, greater than or equal to.

The behavior of each operation is dependent on the type of each evaluated expression.

The comparison semantics for each operator are defined below based on the corresponding JSON type:

Equality Operators

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 sequence of code points. The literal values `true/false/null` are only equal to their own literal values. Two JSON objects are equal if they have the same set of keys and values (given two JSON objeccts `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]`).

Ordering Operators

Ordering operators `>`, `>=`, `<`, `<=` are **only** valid for numbers. Evaluating any other type with a comparison operator will yield a `null` value, which will result in the element being excluded from the result list. For example, given:

```
search('foo[?a<b]', {"foo": [{"a": "char", "b": "char"},
                             {"a": 2, "b": 1},
                             {"a": 1, "b": 2}]})
```

The three 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 expression results in `null`, so the first element is not included in 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 evalutes to `true`, so the third element is included in the list. The final result of that expression is `[{"a": 1, "b": 2}]`.

1.10.2 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}]
```

1.11 Functions 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 JMESPath expressions.

1.11.1 Data Types

In order to support functions, a type system is needed. The JSON types are used:

- 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 also an additional type that is not a JSON type that's used in JMESPath functions:

- expression (denoted by `&expression`)

1.11.2 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, the following expression creates an array containing the total number of elements in the `foo` object followed by the value of `foo["bar"]`.

```
foo[].[count(@), bar]
```

JMESPath assumes that all function arguments operate on the current node unless the argument is a `literal` or `number` token. Because of this, an expression such as `@.bar` would be equivalent to just `bar`, so the current node is only allowed as a bare expression.

current-node state

At the start of an expression, the value of the current node is the data being evaluated by the JMESPath expression. As an expression is evaluated, the value the 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.

1.11.3 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 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 JMESPath 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 contraints, functions are provided to convert types to other types (`to_string`, `to_number`) which are defined below. No explicit type conversion happens unless a user specifically uses one of these type conversion functions.

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([].to_number(@), ``["1", "2", "3", "notanumber", true]``) -> [1, 2, 3]
```

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

1.12 Built-in Functions

JMESPath has various built-in functions that operate on different data types, documented below. 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)
```

If a function can accept multiple types for an input value, then the multiple types are separated with `|`. 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 below 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`).

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`.

The first function below, `abs` is discussed in detail to demonstrate the above points. Subsequent function definitions will not include these details for brevity, but the same rules apply.

Note: All string related functions are defined on the basis of Unicode code points; they do not take normalization into account.

1.12.1 abs

```
number abs(number $value)
```

Returns the absolute value of the provided argument. The signature indicates that a number is returned, and that the input argument `$value` **must** resolve to a number, otherwise a `invalid-type` error is triggered.

Below is a worked example. 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. Validate the type of the resolved argument. In this case `-1` is of type `number` so it passes the type check.

3. Call the function with the resolved argument:

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

4. 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. Validate the type of the resolved argument. In this case `"2"` is of type `string` so we immediately indicate that an `invalid-type` error occurred.

As a final example, here is the steps for evaluating `abs(to_number(bar))`:

1. Evaluate the input argument against the current data:

   ```
   search(to_number(bar), {"foo": -1, "bar": "2"})
   ```

2. In order to evaluate the above expression, we need to evaluate `to_number(bar)`:

   ```
   search(bar, {"foo": -1, "bar": "2"}) -> "2"
   # Validate "2" passes the type check for to_number, which it does.
   to_number("2") -> 2
   ```

   Note that to_number is defined below.

3. Now we can evaluate the original expression:

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

4. Call the function with the final resolved value:

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

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

Table 1.1: Examples

Expression	Result
`abs(1)`	1
`abs(-1)`	1
`abs(`abc`)`	`<error: invalid-type>`

1.12.2 avg

```
number avg(array[number] $elements)
```

Returns the average of the elements in the provided array.

An empty array will produce a return value of null.

Table 1.2: Examples

Given	Expression	Result
`[10, 15, 20]`	`avg(@)`	15
`[10, false, 20]`	`avg(@)`	`<error: invalid-type>`
`[false]`	`avg(@)`	`<error: invalid-type>`
`false`	`avg(@)`	`<error: invalid-type>`

1.12.3 contains

```
boolean contains(array|string $subject, any $search)
```

Returns `true` if the given `$subject` contains the provided `$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 the provided `$subject` is a string, this function returns true if the string contains the provided `$search` argument.

Table 1.3: Examples

Given	Expression	Result
n/a	`contains('foobar', 'foo')`	`true`
n/a	`contains('foobar', 'not')`	`false`
n/a	`contains('foobar', 'bar')`	`true`
n/a	`contains('false', 'bar')`	`<error: invalid-type>`
n/a	`contains('foobar', 123)`	`false`
`["a", "b"]`	`contains(@, 'a')`	`true`
`["a"]`	`contains(@, 'a')`	`true`
`["a"]`	`contains(@, 'b')`	`false`

1.12.4 ceil

```
number ceil(number $value)
```

Returns the next highest integer value by rounding up if necessary.

Table 1.4: Examples

Expression	Result
`ceil('1.001')`	2
`ceil('1.9')`	2
`ceil('1')`	1
`ceil('abc')`	`null`

1.12.5 ends_with

```
boolean ends_with(string $subject, string $prefix)
```

Returns `true` if the `$subject` ends with the `$prefix`, otherwise this function returns `false`.

Table 1.5: Examples

Given	Expression	Result
`foobarbaz`	``ends_with(@, ``baz)``	`true`
`foobarbaz`	``ends_with(@, ``foo)``	`false`
`foobarbaz`	``ends_with(@, ``z)``	`true`

1.12.6 floor

```
number floor(number $value)
```

Returns the next lowest integer value by rounding down if necessary.

Table 1.6: Examples

Expression	Result
`floor(`1.001`)`	1
`floor(`1.9`)`	1
`floor(`1`)`	1

1.12.7 join

```
string join(string $glue, array[string] $stringsarray)
```

Returns all of the elements from the provided `$stringsarray` array joined together using the `$glue` argument as a separator between each.

Table 1.7: Examples

Given	Expression	Result
`["a", "b"]`	`join(`, `, @)`	“a, b”
`["a", "b"]`	`join(``, @)“`	“ab”
`["a", false, "b"]`	`join(`, `, @)`	`<error: invalid-type>`
`[false]`	`join(`, `, @)`	`<error: invalid-type>`

1.12.8 keys

```
array keys(object $obj)
```

Returns an array containing the keys of the provided object.

Table 1.8: Examples

Given	Expression	Result
`{"foo": "baz", "bar": "bam"}`	`keys(@)`	`["foo", "bar"]`
`{}`	`keys(@)`	`[]`
`false`	`keys(@)`	`<error: invalid-type>`
`[b, a, c]`	`keys(@)`	`<error: invalid-type>`

1.12.9 length

```
number length(string|array|object $subject)
```

Returns the length of the given argument using the following types rules:

1. string: returns the number of code points in the string
2. array: returns the number of elements in the array
3. object: returns the number of key-value pairs in the object

Table 1.9: Examples

Given	Expression	Result
n/a	`length('abc')`	3
"current"	`length(@)`	7
"current"	`length(not_there)`	`<error: invalid-type>`
`["a", "b", "c"]`	`length(@)`	3
`[]`	`length(@)`	0
`{}`	`length(@)`	0
`{"foo": "bar", "baz": "bam"}`	`length(@)`	2

1.12.10 max

```
number max(array[number]|array[string] $collection)
```

Returns the highest found number in the provided array argument.

An empty array will produce a return value of null.

Table 1.10: Examples

Given	Expression	Result
`[10, 15]`	`max(@)`	15
`["a", "b"]`	`max(@)`	"b"
`["a", 2, "b"]`	`max(@)`	`<error: invalid-type>`
`[10, false, 20]`	`max(@)`	`<error: invalid-type>`

1.12.11 max_by

```
max_by(array elements, expression->number|expression->string expr)
```

Return the maximum element in an array using the expression `expr` as the comparison key. The entire maximum element is returned. Below are several examples using the `people` array (defined above) as the given input.

Table 1.11: Examples

Expression	Result
`max_by(people, &age)`	`{"age": 50, "age_str": "50", "bool": false, "name": "d"}`
`max_by(people, &age).age`	50
`max_by(people, &to_number(age_str))`	`{"age": 50, "age_str": "50", "bool": false, "name": "d"}`
`max_by(people, &age_str)`	<error: invalid-type>
`max_by(people, age)`	<error: invalid-type>

1.12.12 min

```
number min(array[number]|array[string] $collection)
```

Returns the lowest found number in the provided `$collection` argument.

Table 1.12: Examples

Given	Expression	Result
`[10, 15]`	`min(@)`	10
`["a", "b"]`	`min(@)`	"a"
`["a", 2, "b"]`	`min(@)`	`<error: invalid-type>`
`[10, false, 20]`	`min(@)`	`<error: invalid-type>`

1.12.13 min_by

```
min_by(array elements, expression->number|expression->string expr)
```

Return the minimum element in an array using the expression `expr` as the comparison key. The entire maximum element is returned. Below are several examples using the `people` array (defined above) as the given input.

Table 1.13: Examples

Expression	Result
`min_by(people, &age)`	`{"age": 10, "age_str": "10", "bool": true, "name": 3}`
`min_by(people, &age).age`	10
`min_by(people, &to_number(age_str))`	`{"age": 10, "age_str": "10", "bool": true, "name": 3}`
`min_by(people, &age_str)`	`<error: invalid-type>`
`min_by(people, age)`	`<error: invalid-type>`

1.12.14 not_null

```
any not_null(any $argument [, any $...])
```

Returns 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 encounted. If all arguments values resolve to `null`, then a value of `null` is returned.

Table 1.14: Examples

Given	Expression	Result
`{"a": null, "b": null, "c": [], "d": "foo"}`	`not_null(no_exist, a, b, c, d)`	`[]`
`{"a": null, "b": null, "c": [], "d": "foo"}`	`not_null(a, b, 'null', d, c)`	`"foo"`
`{"a": null, "b": null, "c": [], "d": "foo"}`	`not_null(a, b)`	`null`

1.12.15 reverse

```
array reverse(string|array $argument)
```

Reverses the order of the `$argument`.

Table 1.15: Examples

Given	Expression	Result
`[0, 1, 2, 3, 4]`	`reverse(@)`	`[4, 3, 2, 1, 0]`
“[]	`reverse(@)`	`[]`
`["a", "b", "c", 1, 2, 3]`	`reverse(@)`	`[3, 2, 1, "c", "b", "a"]`
`"abcd`	`reverse(@)`	`dcba`

1.12.16 sort

```
array sort(array[number]|array[string] $list)
```

This function accepts an array `$list` argument and returns the sorted elements of the `$list` as an array.

The array must be a list of strings or numbers. Sorting strings is based on code points. Locale is not taken into account.

Table 1.16: Examples

Given	Expression	Result
`[b, a, c]`	`sort(@)`	`[a, b, c]`
`[1, a, c]`	`sort(@)`	`[1, a, c]`
`[false, [], null]`	`sort(@)`	`[[], null, false]`
`[[], {}, false]`	`sort(@)`	`[{}, [], false]`
`{"a": 1, "b": 2}`	`sort(@)`	`null`
`false`	`sort(@)`	`null`

1.12.17 sort_by

```
sort_by(array elements, expression->number|expression->string expr)
```

Sort an array using an expression `expr` as the sort key. Below are several examples using the `people` array (defined above) as the given input. `sort_by` follows the same sorting logic as the `sort` function.

Table 1.17: Examples

Expression	Result
`sort_by(people, &age)[].age`	`[10, 20, 30, 40, 50]`
`sort_by(people, &age)[0]`	`{"age": 10, "age_str": "10", "bool": true, "name": 3}`
`sort_by(people, &to_number(age_str))[0]`	`{"age": 10, "age_str": "10", "bool": true, "name": 3}`

1.12.18 starts_with

```
boolean starts_with(string $subject, string $prefix)
```

Returns `true` if the `$subject` starts with the `$prefix`, otherwise this function returns `false`.

Table 1.18: Examples

Given	Expression	Result
`foobarbaz`	`starts_with(@, ``foo)“`	`true`
`foobarbaz`	`starts_with(@, ``baz)“`	`false`
`foobarbaz`	`starts_with(@, ``f)“`	`true`

1.12.19 sum

```
number sum(array[number] $collection)
```

Returns the sum of the provided array argument.

An empty array will produce a return value of 0.

Table 1.19: Examples

Given	Expression	Result
`[10, 15]`	`sum(@)`	25
`[10, false, 20]`	`max(@)`	`<error: invalid-type>`
`[10, false, 20]`	`sum([].to_number(@))`	30
`[]`	`sum(@)`	0

1.12.20 to_string

```
string to_string(any $arg)
```

- string - Returns the passed in value.
- number/array/object/boolean - The JSON encoded value of the object. The JSON encoder should emit the encoded JSON value without adding any additional new lines.

Table 1.20: Examples

Given	Expression	Result
`null`	`to_string(`2`)`	`"2"`

1.12.21 to_number

```
number to_number(any $arg)
```

- string - Returns the parsed number. Any string that conforms to the `json-number` production is supported. Note that the floating number support will be implementation specific, but implementations should support at least IEEE 754-2008 binary64 (double precision) numbers, as this is generally available and widely used.
- number - Returns the passed in value.
- array - null
- object - null
- boolean - null
- null - null

1.12.22 type

```
string type(array|object|string|number|boolean|null $subject)
```

Returns 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

Table 1.21: Examples

Given	Expression	Result
"foo"	`type(@)`	"string"
`true`	`type(@)`	"boolean"
`false`	`type(@)`	"boolean"
`null`	`type(@)`	"null"
123	`type(@)`	number
123.05	`type(@)`	number
`["abc"]`	`type(@)`	"array"
`{"abc": "123"}`	`type(@)`	"object"

1.12.23 values

```
array values(object $obj)
```

Returns the values of the provided object.

Table 1.22: Examples

Given	Expression	Result
`{"foo": "baz", "bar": "bam"}`	`values(@)`	`["baz", "bam"]`
`["a", "b"]`	`values(@)`	`<error: invalid-type>`
`false`	`values(@)`	`<error: invalid-type>`

1.13 Pipe Expressions

```
pipe-expression  = 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"]
```

1.13.1 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]
```

CHAPTER 2

JMESPath Proposals

This document lists all of the proposed JMESPath syntax and functionality changes. Proposals are marked as either "draft", "accepted", or "rejected".

2.1 Nested Expressions

JEP 1

Author Michael Dowling

Status accepted

Created 27-Nov-2013

2.1.1 Abstract

This document proposes modifying the JMESPath grammar to support arbitrarily nested expressions within `multi-select-list` and `multi-select-hash` expressions.

2.1.2 Motivation

This JMESPath grammar currently does not allow arbitrarily nested expressions within `multi-select-list` and `multi-select-hash` expressions. This prevents nested branching expressions, nested `multi-select-list` expressions within other multi expressions, and nested ‘‘or-expression‘‘s within any multi-expression.

By allowing any expression to be nested within a `multi-select-list` and `multi-select-hash` expression, we can trim down several grammar rules and provide customers with a much more flexible expression DSL.

Supporting arbitrarily nested expressions within other expressions requires:

- Updating the grammar to remove `non-branched-expr`
- Updating compliance tests to add various permutations of the grammar to ensure implementations are compliant.
- Updating the JMESPath documentation to reflect the ability to arbitrarily nest expressions.

2.1.3 Nested Expression Examples

Nested branch expressions

Given:

```
{
    "foo": {
        "baz": [
            {
                "bar": "abc"
            }, {
                "bar": "def"
            }
        ],
        "qux": ["zero"]
    }
}
```

With: `foo.[baz[*].bar, qux[0]]`

Result:

```
[
    [
        "abc",
        "def"
    ],
    "zero"
]
```

Nested branch expressions with nested mutli-select

Given:

```
{
    "foo": {
        "baz": [
            {
                "bar": "a",
                "bam": "b",
                "boo": "c"
            }, {
                "bar": "d",
                "bam": "e",
                "boo": "f"
            }
        ],
        "qux": ["zero"]
    }
}
```

With: `foo.[baz[*].[bar, boo], qux[0]]`

Result:

```
[
    [
        [
```

```
            "a",
            "c"
        ],
        [
            "d",
            "f"
        ]
    ],
    "zero"
]
```

Nested or expressions

Given:

```
{
    "foo": {
        "baz": [
            {
                "bar": "a",
                "bam": "b",
                "boo": "c"
            }, {
                "bar": "d",
                "bam": "e",
                "boo": "f"
            }
        ],
        "qux": ["zero"]
    }
}
```

With: `foo.[baz[*].not_there || baz[*].bar, qux[0]]`

Result:

```
[
    [
        "a",
        "d"
    ],
    "zero"
]
```

No breaking changes

Because there are no breaking changes from this modification, existing multi-select expressions will still work unchanged:

Given:

```
{
    "foo": {
        "baz": {
            "abc": 123,
            "bar": 456
        }
```

```
    }
}
```

With: `foo.[baz, baz.bar]`

Result:

```
[
    {
        "abc": 123,
        "bar": 456
    },
    456
]
```

2.1.4 Modified Grammar

The following modified JMESPath grammar supports arbitrarily nested expressions and is specified using ABNF, as described in RFC4234

```
expression        = sub-expression / index-expression / or-expression / identifier / "*"
expression        =/ multi-select-list / multi-select-hash
sub-expression    = expression "." expression
or-expression     = expression "||" expression
index-expression  = expression bracket-specifier / bracket-specifier
multi-select-list = "[" ( expression *( "," expression ) ) "]"
multi-select-hash = "{" ( keyval-expr *( "," keyval-expr ) ) "}"
keyval-expr       = identifier ":" expression
bracket-specifier = "[" (number / "*") "]"
number            = [-]1*digit
digit             = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "0"
identifier        = 1*char
identifier        =/ quote 1*(unescaped-char / escaped-quote) quote
escaped-quote     = escape quote
unescaped-char    = %x30-10FFFF
escape            = %x5C   ; Back slash: \
quote             = %x22   ; Double quote: '"'
char              = %x30-39 / ; 0-9
                    %x41-5A / ; A-Z
                    %x5F /    ; _
                    %x61-7A / ; a-z
                    %x7F-10FFFF
```

2.2 Improved Identifiers

JEP 6

Author James Saryerwinnie

Status draft

Created 14-Dec-2013

Last Updated 15-Dec-2013

2.2.1 Abstract

This JEP proposes grammar modifications to JMESPath in order to improve identifiers used in JMESPath. In doing so, several inconsistencies in the identifier grammar rules will be fixed, along with an improved grammar for specifying unicode identifiers in a way that is consistent with JSON strings.

2.2.2 Motivation

There are two ways to currently specify an identifier, the unquoted rule:

```
identifier          = 1*char
```

and the quoted rule:

```
identifier          =/ quote 1*(unescaped-char / escaped-quote) quote
```

The `char` rule contains a set of characters that do **not** have to be quoted:

```
char                = %x30-39 / ; 0-9
                      %x41-5A / ; A-Z
                      %x5F /    ; _
                      %x61-7A / ; a-z
                      %x7F-10FFFF
```

There is an ambiguity between the `%x30-39` rule and the `number` rule:

```
number              = ["-"]1*digit
digit               = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "0"
```

It's ambiguous which rule to use. Given a string "123", it's not clear whether this should be parsed as an identifier or a number. Existing implementations **aren't** following this rule (because it's ambiguous) so the grammar should be updated to remove the ambiguity, specifically, an unquoted identifier can only start with the characters `[a-zA-Z_]`.

Unicode

JMESPath supports unicode through the `char` and `unescaped-char` rule:

```
unescaped-char      = %x30-10FFFF
char                = %x30-39 / ; 0-9
                      %x41-5A / ; A-Z
                      %x5F /    ; _
                      %x61-7A / ; a-z
                      %x7F-10FFFF
```

However, JSON supports a syntax for escaping unicode characters. Any character in the Basic Multilingual Plane (BMP) can be escaped with:

```
char = escape (%x75 4HEXDIG )  ; \uXXXX
```

Similar to the way that XPath supports numeric character references used in XML (`&#nnnn`), JMESPath should support the same escape sequences used in JSON. JSON also supports a 12 character escape sequence for characters outside of the BMP, by encoding the UTF-16 surrogate pair. For example, the code point `U+1D11E` can be represented as `"\uD834\uDD1E"`.

Escape Sequences

Consider the following JSON object:

```
{"foo\nbar": "baz"}
```

A JMESPath expression should be able to retrieve the value of baz. With the current grammar, one must rely on the environment's ability to input control characters such as the newline (`%x0A`). This can be problematic in certain environments. For example, in python, this is not a problem:

```
>>> jmespath_expression = "foo\nbar"
```

Python will interpret the sequence `"\n"` (`%x5C %x6E`) as the newline character `%x0A`. However, consider Bash:

```
$ foo --jmespath-expression "foo\nbar"
```

In this situation, bash will not interpret the `"\n"` (`%x5C %x6E`) sequence.

2.2.3 Specification

The `char` rule contains a set of characters that do **not** have to be quoted. The new set of characters that do not have to quoted will be:

```
unquoted-string   = (%x41-5A / %x61-7A / %x5F) *(%x30-39 / %x41-5A / %x5F / %x61-7A)
```

In order for an identifier to not be quoted, it must start with `[A-Za-z_]`, then must be followed by zero or more `[0-9A-Za-z_]`.

The unquoted rule is updated to account for all JSON supported escape sequences:

```
quoted-string     =/ quote 1*(unescaped-char / escaped-char) quote
```

The full rule for an identifier is:

```
identifier        = unquoted-string / quoted-string
unquoted-string   = (%x41-5A / %x61-7A / %x5F) *(  ; a-zA-Z_
                      %x30-39  /  ; 0-9
                      %x41-5A /  ; A-Z
                      %x5F    /  ; _
                      %x61-7A)   ; a-z
quoted-string     = quote 1*(unescaped-char / escaped-char) quote
unescaped-char    = %x20-21 / %x23-5B / %x5D-10FFFF
escape            = %x5C   ; Back slash: \
quote             = %x22   ; Double quote: '"'
escaped-char      = escape (
                      %x22 /          ; "    quotation mark  U+0022
                      %x5C /          ; \    reverse solidus U+005C
                      %x2F /          ; /    solidus         U+002F
                      %x62 /          ; b    backspace       U+0008
                      %x66 /          ; f    form feed       U+000C
                      %x6E /          ; n    line feed       U+000A
                      %x72 /          ; r    carriage return U+000D
                      %x74 /          ; t    tab             U+0009
                      %x75 4HEXDIG )  ; uXXXX                U+XXXX
```

2.2.4 Rationale

Adopting the same string rules as JSON strings will allow users familiar with JSON semantics to understand how JMESPath identifiers will work.

This change also provides a nice consistency for the literal syntax proposed in JEP 3. With this model, the supported literal strings can be the same as quoted identifiers.

This also will allow the grammar to grow in a consistent way if JMESPath adds support for filtering based on literal values. For example (note that this is just a suggested syntax, not a formal proposal), given the data:

```
{"foo": [{"": ""}, {"": ""}]}
```

You can now have the following JMESPath expressions:

```
foo[?"" = ``]
foo[?"\u2713" = `\u2713`]
```

As a general property, any supported JSON string is now a supported quoted identifier.

2.2.5 Impact

For any implementation that was parsing digits as an identifier, identifiers starting with digits will no longer be valid, e.g. `foo.0.1.2`.

There are several compliance tests that will have to be updated as a result of this JEP. They were arguably wrong to begin with.

basic.json

The following needs to be changed because identifiers starting with a number must now be quoted:

```
-             "expression": "foo.1",
+             "expression": "foo.\"1\"",
              "result": ["one", "two", "three"]
          },
          {
-             "expression": "foo.1[0]",
+             "expression": "foo.\"1\"[0]",
              "result": "one"
          },
```

Similarly, the following needs to be changed because an unquoted identifier cannot start with `-`:

```
-             "expression": "foo.-1",
+             "expression": "foo.\"-1\"",
              "result": "bar"
          }
```

escape.json

The escape.json has several more interseting cases that need to be updated. This has to do with the updated escaping rules. Each one will be explained.

```
-            "expression": "\"foo\nbar\"",
+            "expression": "\"foo\\nbar\"",
             "result": "newline"
          },
```

This has to be updated because a JSON parser will interpret the `\n` sequence as the newline character. The newline character is **not** allowed in a JMESPath identifier (note that the newline character `%0A` is not in any rule). In order for a JSON parser to create a sequence of `%x5C %x6E`, the JSON string must be `\\n` (`%x5C %x5C %x6E`).

```
-            "expression": "\"c:\\\\windows\\path\"",
+            "expression": "\"c:\\\\\\\\windows\\\\path\"",
             "result": "windows"
          },
```

The above example is a more pathological case of escaping. In this example, we have a string that represents a windows path "c:\windowpath". There are two levels of escaping happening here, one at the JSON parser, and one at the JMESPath parser. The JSON parser will take the sequence `"\"c:\\\\\\\\windows\\\\path\""` and create the string `"\"c:\\\\windows\\path\""`. The JMESPath parser will take the string `"\"c:\\\\windows\\path\"'` and, applying its own escaping rules, will look for a key named `c:\\windows\path`.

2.3 Filter Expressions

JEP 7

Author James Saryerwinnie

Status accepted

Created 16-Dec-2013

2.3.1 Abstract

This JEP proposes grammar modifications to JMESPath to allow for filter expressions. A filtered expression allows list elements to be selected based on matching expressions. A literal expression is also introduced (from JEP 3) so that it is possible to match elements against literal values.

2.3.2 Motivation

A common request when querying JSON objects is the ability to select elements based on a specific value. For example, given a JSON object:

```
{"foo": [{"state": "WA", "value": 1},
         {"state": "WA", "value": 2},
         {"state": "CA", "value": 3},
         {"state": "CA", "value": 4}]}
```

A user may want to select all objects in the `foo` list that have a `state` key of `WA`. There is currently no way to do this in JMESPath. This JEP will introduce a syntax that allows this:

```
foo[?state == `WA`]
```

Additionally, a user may want to project additional expressions onto the values matched from a filter expression. For example, given the data above, select the `value` key from all objects that have a `state` of `WA`:

```
foo[?state == `WA`].value
```

would return `[1, 2]`.

2.3.3 Specification

The updated grammar for filter expressions:

```
bracket-specifier      = "[" (number / "*") "]" / "[]"
bracket-specifier      =/ "[?" list-filter-expression "]"
list-filter-expression = expression comparator expression
comparator             = "<" / "<=" / "==" / ">=" / ">" / "!="
expression             =/ literal
literal                = "`" json-value "`"
literal                =/ "`" 1*(unescaped-literal / escaped-literal) "`"
unescaped-literal      = %x20-21 /       ; space !
                         %x23-5A /   ; # - [
                         %x5D-5F /   ; ] ^ _
                         %x61-7A     ; a-z
                         %x7C-10FFFF ; |}~ ...
escaped-literal        = escaped-char / (escape %x60)
```

The `json-value` rule is any valid json value. While it's recommended that implementations use an existing JSON parser to parse the `json-value`, the grammar is added below for completeness:

```
json-value = "false" / "null" / "true" / json-object / json-array /
             json-number / json-quoted-string
json-quoted-string = %x22 1*(unescaped-literal / escaped-literal) %x22
begin-array     = ws %x5B ws  ; [ left square bracket
begin-object    = ws %x7B ws  ; { left curly bracket
end-array       = ws %x5D ws  ; ] right square bracket
end-object      = ws %x7D ws  ; } right curly bracket
name-separator  = ws %x3A ws  ; : colon
value-separator = ws %x2C ws  ; , comma
ws              = *(%x20 /              ; Space
                    %x09 /              ; Horizontal tab
                    %x0A /              ; Line feed or New line
                    %x0D                ; Carriage return
                   )
json-object = begin-object [ member *( value-separator member ) ] end-object
member = quoted-string name-separator json-value
json-array = begin-array [ json-value *( value-separator json-value ) ] end-array
json-number = [ minus ] int [ frac ] [ exp ]
decimal-point = %x2E       ; .
digit1-9 = %x31-39         ; 1-9
e = %x65 / %x45            ; e E
exp = e [ minus / plus ] 1*DIGIT
frac = decimal-point 1*DIGIT
int = zero / ( digit1-9 *DIGIT )
minus = %x2D               ; -
plus = %x2B                ; +
zero = %x30                ; 0
```

Comparison Operators

The following operations are supported:

- ==, tests for equality.
- !=, tests for inequality.
- <, less than.
- <=, less than or equal to.
- >, greater than.
- >=, greater than or equal to.

The behavior of each operation is dependent on the type of each evaluated expression.

The comparison semantics for each operator are defined below based on the corresponding JSON type:

Equality Operators

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 sequence of code points. The literal values `true/false/null` are only equal to their own literal values. Two JSON objects are equal if they have the same set of keys (for each key in the first JSON object there exists a key with equal value in the second JSON object). Two JSON arrays are equal if they have equal elements in the same order (given two arrays `x` and `y`, for each `i` in `x`, `x[i] == y[i]`).

Ordering Operators

Ordering operators `>, >=, <, <=` are **only** valid for numbers. Evaluating any other type with a comparison operator will yield a `null` value, which will result in the element being excluded from the result list. For example, given:

```
search('foo[?a<b]', {"foo": [{"a": "char", "b": "char"},
                             {"a": 2, "b": 1},
                             {"a": 1, "b": 2}]})
```

The three 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 expression results in `null`, so the first element is not included in 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 evalutes to `true`, so the third element is included in the list. The final result of that expression is `[{"a": 1, "b": 2}]`.

Filtering Semantics

When a filter expression is matched, the matched element in its entirety is included in the filtered response.

Using the previous example, given the following data:

```
{"foo": [{"state": "WA", "value": 1},
         {"state": "WA", "value": 2},
         {"state": "CA", "value": 3},
         {"state": "CA", "value": 4}]}
```

The expression `foo[?state == 'WA']` will return the following value:

```
[{"state": "WA", "value": 1}]
```

Literal Expressions

Literal expressions are also added in the JEP, which is essentially a JSON value surrounded by the "`" character. You can escape the "`" character via "\`", and if the character "`" appears in the JSON value, it must also be escaped. A simple two pass algorithm in the lexer could first process any escaped "`" characters before handing the resulting string to a JSON parser.

Because string literals are by far the most common type of JSON value, an alternate syntax is supported where the starting and ending double quotes are not required for strings. For example:

```
`foobar`   -> "foobar"
`"foobar"` -> "foobar"
`123`      -> 123
`"123"`    -> "123"
`123.foo`  -> "123.foo"
`true`     -> true
`"true"`   -> "true"
`truee`    -> "truee"
```

Literal expressions aren't allowed on the right hand side of a subexpression:

```
foo[*].`literal`
```

but they are allowed on the left hand side:

```
`{"foo": "bar"}`.foo
```

They may also be included in other expressions outside of a filter expressions. For example:

```
{value: foo.bar, type: `multi-select-hash`}
```

2.3.4 Rationale

The proposed filter expression syntax was chosen such that there is sufficient expressive power for any type of filter one might need to perform while at the same time being as minimal as possible. To help illustrate this, below are a few alternate syntax that were considered.

In the simplest case where one might filter a key based on a literal value, a possible filter syntax would be:

```
foo[bar == baz]
```

or in general terms: `[identifier comparator literal-value]`. However this has several issues:

- It is not possible to filter based on two expressions (get all elements whose `foo` key equals its `bar` key.
- The literal value is on the right hand side, making it hard to troubleshoot if the identifier and literal value are swapped: `foo[baz == bar]`.
- Without some identifying token unary filters would not be possible as they would be ambiguous. Is the expression `[foo]` filtering all elements with a foo key with a truth value or is it a multiselect-list selecting the `foo` key from each hash? Starting a filter expression with a token such as `[?` make it clear that this is a filter expression.
- This makes the syntax for filtering against literal JSON arrays and objects hard to visually parse. "Filter all elements whose `foo` key is a single list with a single integer value of 2: `[foo == [2]]`.
- Adding literal expressions makes them useful even outside of a filter expression. For example, in a `multi-select-hash`, you can create arbitrary key value pairs: `{a: foo.bar, b: `some string`}`.

This JEP is purposefully minimal. There are several extensions that can be added in future:

- Support any arbitrary expression within the `[? ... ]`. This would enable constructs such as or expressions within a filter. This would allow unary expressions.

In order for this to be useful we need to define what corresponds to true and false values, e.g. an empty list is a false value. Additionally, "or expressions" would need to change its semantics to branch based on the true/false value of an expression instead of whether or not the expression evalutes to null.

This is certainly a direction to take in the future, adding arbitrary expressions in a filter would be a backwards compatible change, so it's not part of this JEP.

- Allow filter expressions as top level expressions. This would potentially just return `true/false` for any value that it matched.

This might be useful if you can combine this with something that can accept a list to use as a mask for filtering other elements.

2.4 Pipe Expressions

JEP 4

Author Michael Dowling

Status accepted

Created 07-Dec-2013

2.4.1 Abstract

This document proposes adding support for piping expressions into subsequent expressions.

2.4.2 Motivation

The current JMESPath grammar allows for projections at various points in an expression. However, it is not currently possible to operate on the result of a projection as a list.

The following example illustrates that it is not possible to operate on the result of a projection (e.g., take the first match of a projection).

Given:

```
{
    "foo": {
        "a": {
            "bar": [1, 2, 3]
        },
        "b": {
            "bar": [4, 5, 6]
        }
    }
}
```

Expression:

```
foo.*.bar[0]
```

The result would be element 0 of each `bar`:

```
[1, 4]
```

With the addition of filters, we could pass the result of one expression to another, operating on the result of a projection (or any expression).

Expression:

```
foo.*.bar | [0]
```

Result:

```
[1, 2, 3]
```

Not only does this give us the ability to operate on the result of a projection, but pipe expressions can also be useful for breaking down a complex expression into smaller, easier to comprehend, parts.

2.4.3 Modified Grammar

The following modified JMESPath grammar supports piped expressions.

```
expression        = sub-expression / index-expression / or-expression / identifier / "*"
expression        =/ multi-select-list / multi-select-hash / pipe-expression
sub-expression    = expression "." expression
pipe-expression   = expression "|" expression
or-expression     = expression "||" expression
index-expression  = expression bracket-specifier / bracket-specifier
multi-select-list = "[" ( expression *( "," expression ) ) "]"
multi-select-hash = "{" ( keyval-expr *( "," keyval-expr ) ) "}"
keyval-expr       = identifier ":" expression
bracket-specifier = "[" (number / "*") "]" / "[]"
number            = [-]1*digit
digit             = "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9" / "0"
identifier        = 1*char
identifier        =/ quote 1*(unescaped-char / escaped-quote) quote
escaped-quote     = escape quote
unescaped-char    = %x30-10FFFF
escape            = %x5C   ; Back slash: \
quote             = %x22   ; Double quote: '"'
char              = %x30-39 / ; 0-9
                    %x41-5A / ; A-Z
                    %x5F /    ; _
                    %x61-7A / ; a-z
                    %x7F-10FFFF
```

Note: `pipe-expression` has a higher precendent than the `or-operator`

2.4.4 Compliance Tests

```
[{
  "given": {
    "foo": {
      "bar": {
        "baz": "one"
      },
      "other": {
```

```
        "baz": "two"
      },
      "other2": {
        "baz": "three"
      },
      "other3": {
        "notbaz": ["a", "b", "c"]
      },
      "other4": {
        "notbaz": ["d", "e", "f"]
      }
    }
  },
  "cases": [
    {
      "expression": "foo.*.baz | [0]",
      "result": "one"
    },
    {
      "expression": "foo.*.baz | [1]",
      "result": "two"
    },
    {
      "expression": "foo.*.baz | [2]",
      "result": "three"
    },
    {
      "expression": "foo.bar.* | [0]",
      "result": "one"
    },
    {
      "expression": "foo.*.notbaz | [*]",
      "result": [["a", "b", "c"], ["d", "e", "f"]]
    },
    {
      "expression": "foo | bar",
      "result": {"baz": "one"}
    },
    {
      "expression": "foo | bar | baz",
      "result": "one"
    },
    {
      "expression": "foo|bar| baz",
      "result": "one"
    },
    {
      "expression": "not_there | [0]",
      "result": null
    },
    {
      "expression": "not_there | [0]",
      "result": null
    },
    {
      "expression": "[foo.bar, foo.other] | [0]",
      "result": {"baz": "one"}
    },
```

```
    {
      "expression": "{\"a\": foo.bar, \"b\": foo.other} | a",
      "result": {"baz": "one"}
    },
    {
      "expression": "{\"a\": foo.bar, \"b\": foo.other} | b",
      "result": {"baz": "two"}
    },
    {
      "expression": "{\"a\": foo.bar, \"b\": foo.other} | *.baz",
      "result": ["one", "two"]
    },
    {
      "expression": "foo.bam || foo.bar | baz",
      "result": "one"
    },
    {
      "expression": "foo | not_there || bar",
      "result": {"baz": "one"}
    }
  ]
}]
```

2.5 Functions

JEP 3

Author Michael Dowling, James Saryerwinnie

Status Draft

Created 27-Nov-2013

2.5.1 Abstract

This document proposes modifying the JMESPath grammar to support function expressions.

2.5.2 Motivation

Functions allow users to easily transform and filter data in JMESPath expressions. As JMESPath is currently implemented, functions would be very useful in `multi-select-list` and `multi-select-hash` expressions to format the output of an expression to contain data that might not have been in the original JSON input. Combined with filtered expressions, functions would be a powerful mechanism to perform any kind of special comparisons for things like `length()`, `contains()`, etc.

2.5.3 Data Types

In order to support functions, a type system is needed. The JSON types are used:

- 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

2.5.4 Syntax Changes

Functions are defined in the `function-expression` rule below. A function expression is an `expression` itself, and is valid any place an `expression` is allowed.

The grammar will require the following grammar additions:

```
function-expression = identifier "(" *(function-arg *("," function-arg ) ) ")"
function-arg        = expression / number / current-node
current-node        = "@"
```

`expression` will need to be updated to add the `function-expression` production:

```
expression        = sub-expression / index-expression / or-expression / identifier / "*"
expression        =/ multi-select-list / multi-select-hash
expression        =/ literal / function-expression
```

A function can accept any number of arguments, and each argument can be an expression. Each function must define a signature that specifies the number and allowed types of its expected arguments. Functions can be variadic.

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, the following expression creates an array containing the total number of elements in the `foo` object followed by the value of `foo["bar"]`.

```
foo[].[count(@), bar]
```

JMESPath assumes that all function arguments operate on the current node unless the argument is a `literal` or `number` token. Because of this, an expression such as `@.bar` would be equivalent to just `bar`, so the current node is only allowed as a bare expression.

current-node state

At the start of an expression, the value of the current node is the data being evaluated by the JMESPath expression. As an expression is evaluated, the value the 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.

2.5.5 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 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 JMESPath 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 contraints, functions are provided to convert types to other types (`to_string`, `to_number`) which are defined below. No explicit type conversion happens unless a user specifically uses one of these type conversion functions.

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([].to_number(@), ``["1", "2", "3", "notanumber", true]``) -> [1, 2, 3]
```

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

2.5.6 Built-in Functions

JMESPath has various built-in functions that operate on different data types, documented below. 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)
```

If a function can accept multiple types for an input value, then the multiple types are separated with `|`. 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 below 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`).

The first function below, `abs` is discussed in detail to demonstrate the above points. Subsequent function definitions will not include these details for brevity, but the same rules apply.

Note: All string related functions are defined on the basis of Unicode code points; they do not take normalization into account.

---

abs

```
number abs(number $value)
```

Returns the absolute value of the provided argument. The signature indicates that a number is returned, and that the input argument `$value` **must** resolve to a number, otherwise a `invalid-type` error is triggered.

Below is a worked example. Given:

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

Evaluating `abs(foo)` works as follows:

1. Evaluate the input argument against the current data:

   ```
   search(foo, {"foo": -11, "bar": 2"}) -> -1
   ```

2. Validate the type of the resolved argument. In this case `-1` is of type `number` so it passes the type check.
3. Call the function with the resolved argument:

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

4. **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(foo, {"foo": -1, "bar": 2"}) -> "2"
   ```

2. Validate the type of the resolved argument. In this case `"2` is of type `string` so the immediate indicate that an `invalid-type` error occurred.

As a final example, here is the steps for evaluating `abs(to_number(bar))`:

1. Evaluate the input argument against the current data:

   ```
   search(to_number(bar), {"foo": -1, "bar": "2"})
   ```

2. In order to evaluate the above expression, we need to evaluate `to_number(bar)`:

   ```
   search(bar, {"foo": -1, "bar": "2"}) -> "2"
   # Validate "2" passes the type check for to_number, which it does.
   to_number("2") -> 2
   ```

3. Now we can evaluate the original expression:

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

4. Call the function with the final resolved value:

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

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

Table 2.1: Examples

Expression	Result
`abs(1)`	1
`abs(-1)`	1
`abs('abc')`	`<error: invalid-type>`

avg

```
number avg(array[number] $elements)
```

Returns the average of the elements in the provided array.

An empty array will produce a return value of null.

Table 2.2: Examples

Given	Expression	Result
`[10, 15, 20]`	`avg(@)`	15
`[10, false, 20]`	`avg(@)`	`<error: invalid-type>`
`[false]`	`avg(@)`	`<error: invalid-type>`
`false`	`avg(@)`	`<error: invalid-type>`

contains

```
boolean contains(array|string $subject, array|object|string|number|boolean $search)
```

Returns `true` if the given `$subject` contains the provided `$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 the provided `$subject` is a string, this function returns true if the string contains the provided `$search` argument.

Table 2.3: Examples

Given	Expression	Result
n/a	`contains('foobar', 'foo')`	`true`
n/a	`contains('foobar', 'not')`	`false`
n/a	`contains('foobar', 'bar')`	`true`
n/a	`contains('false', 'bar')`	`<error: invalid-type>`
n/a	`contains('foobar', 123)`	`false`
`["a", "b"]`	`contains(@, 'a')`	`true`
`["a"]`	`contains(@, 'a')`	`true`
`["a"]`	`contains(@, 'b')`	`false`

ceil

```
number ceil(number $value)
```

Returns the next highest integer value by rounding up if necessary.

Table 2.4: Examples

Expression	Result
`ceil('1.001')`	2
`ceil('1.9')`	2
`ceil('1')`	1
`ceil('abc')`	`null`

floor

```
number floor(number $value)
```

Returns the next lowest integer value by rounding down if necessary.

Table 2.5: Examples

Expression	Result
`floor(`1.001`)`	1
`floor(`1.9`)`	1
`floor(`1`)`	1

join

```
string join(string $glue, array[string] $stringsarray)
```

Returns all of the elements from the provided `$stringsarray` array joined together using the `$glue` argument as a separator between each.

Table 2.6: Examples

Given	Expression	Result
`["a", "b"]`	`join(`, `, @)`	"a, b"
`["a", "b"]`	`join(, @)``	"ab"
`["a", false, "b"]`	`join(`, `, @)`	`<error: invalid-type>`
`[false]`	`join(`, `, @)`	`<error: invalid-type>`

keys

```
array keys(object $obj)
```

Returns an array containing the keys of the provided object.

Table 2.7: Examples

Given	Expression	Result
`{"foo": "baz", "bar": "bam"}`	`keys(@)`	`["foo", "bar"]`
`{}`	`keys(@)`	`[]`
`false`	`keys(@)`	`<error: invalid-type>`
`[b, a, c]`	`keys(@)`	`<error: invalid-type>`

length

```
number length(string|array|object $subject)
```

Returns the length of the given argument using the following types rules:

1. string: returns the number of code points in the string
2. array: returns the number of elements in the array
3. object: returns the number of key-value pairs in the object

Table 2.8: Examples

Given	Expression	Result
n/a	`length('abc')`	3
"current"	`length(@)`	7
"current"	`length(not_there)`	`<error: invalid-type>`
`["a", "b", "c"]`	`length(@)`	3
`[]`	`length(@)`	0
`{}`	`length(@)`	0
`{"foo": "bar", "baz": "bam"}`	`length(@)`	2

max

```
number max(array[number] $collection)
```

Returns the highest found number in the provided array argument.

An empty array will produce a return value of null.

Table 2.9: Examples

Given	Expression	Result
`[10, 15]`	`max(@)`	15
`[10, false, 20]`	`max(@)`	`<error: invalid-type>`

min

```
number min(array[number] $collection)
```

Returns the lowest found number in the provided `$collection` argument.

Table 2.10: Examples

Given	Expression	Result
`[10, 15]`	`min(@)`	10
`[10, false, 20]`	`min(@)`	`<error: invalid-type>`

sort

```
array sort(array $list)
```

This function accepts an array `$list` argument and returns the sorted elements of the `$list` as an array.

The array must be a list of strings or numbers. Sorting strings is based on code points. Locale is not taken into account.

Table 2.11: Examples

Given	Expression	Result
`[b, a, c]`	`sort(@)`	`[a, b, c]`
`[1, a, c]`	`sort(@)`	`[1, a, c]`
`[false, [], null]`	`sort(@)`	`[[], null, false]`
`[[], {}, false]`	`sort(@)`	`[{}, [], false]`
`{"a": 1, "b": 2}`	`sort(@)`	`null`
`false`	`sort(@)`	`null`

to_string

```
string to_string(string|number|array|object|boolean $arg)
```

- string - Returns the passed in value.
- number/array/object/boolean - The JSON encoded value of the object. The JSON encoder should emit the encoded JSON value without adding any additional new lines.

Table 2.12: Examples

Given	Expression	Result
`null`	`to_string('2')`	`"2"`

to_number

```
number to_number(string|number $arg)
```

- string - Returns the parsed number. Any string that conforms to the `json-number` production is supported.
- number - Returns the passed in value.
- array - null
- object - null
- boolean - null

type

```
string type(array|object|string|number|boolean|null $subject)
```

Returns 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

Table 2.13: Examples

Given	Expression	Result
"foo"	`type(@)`	"string"
`true`	`type(@)`	"boolean"
`false`	`type(@)`	"boolean"
`null`	`type(@)`	"null"
123	`type(@)`	number
123.05	`type(@)`	number
`["abc"]`	`type(@)`	"array"
`{"abc": "123"}`	`type(@)`	"object"

values

```
array values(object $obj)
```

Returns the values of the provided object.

Table 2.14: Examples

Given	Expression	Result
`{"foo": "baz", "bar": "bam"}`	`values(@)`	`["baz", "bam"]`
`["a", "b"]`	`values(@)`	`<error: invalid-type>`
`false`	`values(@)`	`<error: invalid-type>`

2.5.7 Compliance Tests

A `functions.json` will be added to the compliance test suite. The test suite will add the following new error types:

- unknown-function
- invalid-arity
- invalid-type

The compliance does not specify **when** the errors are raised, as this will depend on implementation details. For an implementation to be compliant they need to indicate that an error occurred while attempting to evaluate the JMESPath expression.

2.5.8 History

- This JEP originally proposed the literal syntax. The literal portion of this JEP was removed and added instead to JEP 7.
- This JEP originally specified that types matches should return null. This has been updated to specify that an invalid type error should occur instead.

2.6 Expression Types

JEP 8

Author James Saryerwinnie

Status accepted

Created 02-Mar-2013

2.6.1 Abstract

This JEP proposes grammar modifications to JMESPath to allow for expression references within functions. This allows for functions such as `sort_by`, `max_by`, `min_by`. These functions take an argument that resolves to an expression type. This enables functionality such as sorting an array based on an expression that is evaluated against every array element.

2.6.2 Motivation

A useful feature that is common in other expression languages is the ability to sort a JSON object based on a particular key. For example, given a JSON object:

```
{
  "people": [
        {"age": 20, "age_str": "20", "bool": true, "name": "a", "extra": "foo"},
        {"age": 40, "age_str": "40", "bool": false, "name": "b", "extra": "bar"},
        {"age": 30, "age_str": "30", "bool": true, "name": "c"},
        {"age": 50, "age_str": "50", "bool": false, "name": "d"},
        {"age": 10, "age_str": "10", "bool": true, "name": 3}
  ]
}
```

It is not currently possible to sort the `people` array by the `age` key. Also, `sort` is not defined for the `object` type, so it's not currently possible to even sort the `people` array. In order to sort the `people` array, we need to know what key to use when sorting the array.

This concept of sorting based on a key can be generalized. Instead of requiring a key name, an expression can be provided that each element would be evaluated against. In the simplest case, this expression would just be an `identifier`, but more complex expressions could be used such as `foo.bar.baz`.

A simple way to accomplish this might be to create a function like this:

```
sort_by(array arg1, expression)

# Called like:

sort_by(people, age)
sort_by(people, to_number(age_str))
```

However, there's a problem with the `sort_by` function as defined above. If we follow the function argument resolution process we get:

```
sort_by(people, age)

# 1. resolve people
arg1 = search(people, <input data>) -> [{"age": ...}, {...}]

# 2. resolve age
arg2 = search(age, <input data>) -> null

sort_by([{"age": ...}, {...}], null)
```

The second argument is evaluated against the current node and the expression `age` will resolve to `null` because the input data has no `age` key. There needs to be some way to specify that an expression should evaluate to an expression type:

```
arg = search(<some expression>, <input data>) -> <expression: age>
```

Then the function definition of `sort_by` would be:

```
sort_by(array arg1, expression arg2)
```

2.6.3 Specification

The following grammar rules will be updated to:

```
function-arg        = expression /
                      current-node /
                      "&" expression
```

Evaluating an expression reference should return an object of type "expression". The list of data types supported by a function will now be:

- 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`)

Function signatures can now be specified using this new `expression` type. Additionally, a function signature can specify the return type of the expression. 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.

Note that any valid expression is allowed after `&`, so the following expressions are valid:

```
sort_by(people, &foo.bar.baz)
sort_by(people, &foo.bar[0].baz)
sort_by(people, &to_number(foo[0].bar))
```

Additional Functions

The following functions will be added:

sort_by

```
sort_by(array elements, expression->number|expression->string expr)
```

Sort an array using an expression `expr` as the sort key. Below are several examples using the `people` array (defined above) as the given input. `sort_by` follows the same sorting logic as the `sort` function.

Table 2.15: Examples

Expression	Result
`sort_by(people, &age)[].age`	[10, 20, 30, 40, 50]
`sort_by(people, &age)[0]`	{"age": 10, "age_str": "10", "bool": true, "name": 3}
`sort_by(people, &to_number(age_str))[0]`	{"age": 10, "age_str": "10", "bool": true, "name": 3}

max_by

```
max_by(array elements, expression->number expr)
```

Return the maximum element in an array using the expression `expr` as the comparison key. The entire maximum element is returned. Below are several examples using the `people` array (defined above) as the given input.

Table 2.16: Examples

Expression	Result
`max_by(people, &age)`	{“age”: 50, “age_str”: “50”, “bool”: false, “name”: “d”},
`max_by(people, &age).age`	50
`max_by(people, &to_number(age_str))`	{“age”: 50, “age_str”: “50”, “bool”: false, “name”: “d”},
`max_by(people, &age_str)`	<error: invalid-type>
`max_by(people, age)`	<error: invalid-type>

min_by

```
min_by(array elements, expression->number expr)
```

Return the minimum element in an array using the expression `expr` as the comparison key. The entire maximum element is returned. Below are several examples using the `people` array (defined above) as the given input.

Table 2.17: Examples

Expression	Result
`min_by(people, &age)`	{“age”: 10, “age_str”: “10”, “bool”: true, “name”: 3}
`min_by(people, &age).age`	10
`min_by(people, &to_number(age_str))`	{“age”: 10, “age_str”: “10”, “bool”: true, “name”: 3}
`min_by(people, &age_str)`	<error: invalid-type>
`min_by(people, age)`	<error: invalid-type>

Alternatives

There were a number of alternative proposals considered. Below outlines several of these alternatives.

Logic in Argument Resolver

The first proposed choice (which was originally in JEP-3 but later removed) was to not have any syntactic construct for specifying functions, and to allow the function signature to dictate whether or not an argument was resolved. The signature for `sort_by` would be:

```
sort_by(array arg1, any arg2)
arg1 -> resolved
arg2 -> not resolved
```

Then the argument resolver would introspect the argument specification of a function to determine what to do. Roughly speaking, the pseudocode would be:

```
call-function(current-data)
arglist = []
for each argspec in functions-argspec:
    if argspect.should_resolve:
      arglist <- resolve(argument, current-data)
    else
      arglist <- argument
type-check(arglist)
return invoke-function(arglist)
```

However, there are several reasons not to do this:

- This imposes a specific implementation. This implementation would be challenging in a bytecode VM, as the CALL bytecode will typically resolve arguments onto the stack and allow the function to then pop arguments off the stack and perform its own arity validation.
- This deviates from the "standard" model of how functions are traditionally implemented.

Specifying Expressions as Strings

Another proposed alternative was to allow the expression to be a string type and to give functions the capability to parse/eval expressions. The `sort_by` function would look like this:

```
sort_by(people, 'age')
sort_by(people, 'foo.bar.baz')
```

The main reasons this proposal was not chosen was because:

- This complicates the implementations. For implementations that walk the AST inline, this means AST nodes need access to the parser. For external tree visitors, the visitor needs access to the parser.
- This moves what *could* by a compile time error into a run time error. The evaluation of the expression string happens when the function is invoked.

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*