JSONPath

June 15th (Tuesday), 09:00-11:00 UTC
(11:00–13:00 CEST, 02:00–04:00 PDT)

#97

Addresses issue #84 (terminology)

Most suggested changes were picked for #101

Remaining issue: Values vs. Literals

JSON RFC (RFC 8259)

Describes **JSON text**.

Discusses **JSON values**,
but never really what is different between
the value itself and its representation

`10, 1e1, 100e-1, 10.0, 1.0e1, 1.00e1, 10.00`
are all the same **number**
in different notation ("number **literal**")

Literals in JSONPath

JSONPath syntax denotes **operators**, punctuation, and **values**

We need to explain where JSONPath literals for values are
the same as in JSON (e.g., numbers, false, true, null),
and where we have our own literal syntax (strings).

JSONPath strings (the **values**):
sequence of one or more Unicode codepoints.

JSON string literals

JSON string literals:
Always double-quoted.
No C0 characters.
Can backslash-escape `"` `\` `/` `b` `f` `n` `r` `t` and uXXXX; nothing else.

```
> a = "\"\\'\""
> console.log(a)
"\'"
> JSON.parse(a)
Uncaught SyntaxError: Unexpected token ' in JSON at position 2
```

JavaScript string literals: way more permissive. Irrelevant.

PR

```
String:
 : A sequence of UTF-8 characters surrounded by either single quotes (`)
   or double quotes (").
   Escaping quote characters is required only for the specific character
   which surrounds the string.
   For example, a double quote character (") within a string surrounded
   by single quotes (') does not need to be escaped.
   Single quotes are preferred.
```

JSONPath string literals (#97)

JSONPath string literals:
almost, but not entirely unlike JSON string literals.

— allow outer single quotes and their escaping?

— what else?

What else?

Any other literals that differ from their JSON counterparts?

Any literals for data types that aren't in JSON?

- JSONPath nodes:
  we already have the »**$**« literal/expression/…
- JSONPath regexps?
- What else?

#99 (#98)

#98: Stefan's selector PR
#99: Fixes so that it builds; branch name ≠ `main`

Discussion happens under both PRs (sorry about that)
#99 discussion mostly editorial; can ignore today

#98 surfaces undecided questions

#98 really has the **expression language** as a prerequisite.

How powerful should the expression language be?

— should it have arithmetic operations (`+ - * / %`)?
— should it have function calls?
— should there be literals (or constructor notations) for structured values?

What is our stance on implicit conversions?
(Emerging consensus was: No implicit conversions.)
What about conversion to Boolean ("truthy")?

```
quoted-member-name  =  %x22 *double-quoted %x22 /       ; "string"
                       %x27 *single-quoted %x27         ; 'string'

double-quoted       = unescaped /
                      %x27      /                       ; '
                      ESC %x22  /                       ; \"
                      ESC escapable

single-quoted       = unescaped /
                      %x22      /                       ; "
                      ESC %x27  /                       ; \'
                      ESC escapable

ESC                 = %x5C                              ; \  backslash

unescaped           = %x20-21 /                         ; s. RFC 8259
                      %x23-26 /                         ; omit "
                      %x28-5B /                         ; omit '
                      %x5D-10FFFF                       ; omit \

escapable           = (
                          b /         ;  BS backspace U+0008
                          t /         ;  HT horizontal tab U+0009
                          n /         ;  LF line feed U+000A
                          f /         ;  FF form feed U+000C
                          r /         ;  CR carriage return U+000D
                          / /         ;  /  slash (solidus)
                          \ /         ;  \  backslash (reverse solidus)
                          u 4HEXDIG   ;  uXXXX      U+XXXX
                      )
HEXDIG              = %x41-46 /           ;  A-F
                      %x61-66             ;  a-f
```

for now, converging to...

```
fix boolean-expr = *logical-expr
fix logical-expr = [neg-op] ["("] comp-expr *[logical-op comp-expr] [")"]
neg-op        = "!"                                    ; not operator
logical-op    = "||" / "&&"                            ; logical operator
comp-expr     = (rel-path-val
                 ) [(comp-op comparable /  ; comparison
                            regex-op regex     /  ; RegEx test
                            in-op container )]    ; containment test
comp-op       = "==" / "!=" /                          ; comparison ...
                "<"  / ">"  /                          ; operators
                "<=" / ">="
regex-op      = "~="                                   ; RegEx match
in-op         = " in "                                 ; in operator
comparable    = number / quoted-string /               ; primitive ...
                true / false / null /                  ; values only
                rel-path-val /                         ; descendant value
                json-path                              ; any value

rel-path-val = "@" *(dot-selector / index-selector)
```

```
boolean-expr = *logical-expr
logical-expr = [neg-op] ["("] comp-expr *[logical-op comp-expr] [")"]
neg-op       = "!"                                   ; not operator
logical-op   = "||" / "&&"                           ; logical operator
comp-expr    = (rel-path-val /
                calc-val) [(comp-op comparable /  ; comparison
                            regex-op regex     /  ; RegEx test
                            in-op container )]    ; containment test
comp-op      = "==" / "!=" /                         ; comparison ...
               "<"  / ">"  /                         ; operators
               "<=" / ">="
regex-op     = "~="                                  ; RegEx match
in-op        = " in "                                ; in operator
comparable   = number / quoted-string /              ; primitive ...
               true / false / null /                 ; values only
               rel-path-val /                        ; descendant value
               calc_val /                            ; calculated value
               json-path                             ; any value

rel-path-val = "@" *(dot-selector / index-selector)
calc_val     = func "(" [rel-path-val / json-path] ")"
func         = "index"
```

```
@.foo in ["a", "b"]
```

```
"a" in @.foo
```

Example: Comparison with structured values

Should comparison with structured values (e.g., @.foo == [1, 2]) be supported?

If it is not supported, should this silently fail or the attempt cause a syntax error (in #99, it causes a syntax error, but then the text says something else).

— Data types: can we even write and pass around `[1, 2]`?

Stefan's topics: Terminology

RFC 8259 on types of JSON values:

```
   JSON can represent four primitive types (strings, numbers, booleans,
   and null) and two structured types (objects and arrays).
```

So, with respect to types imported from JSON, we have

- primitive (as opposed to atomic/simple), and
- structured (as opposed to complex/container).

(Note that "container" is useful to name the container itself, as opposed to including what's in there.)

Stefan's topics: Terminology for Selectors

- Fundamental point is the alternate wording in text … :
  - … selects a value.
  - … selects a node.

Both is possible with JSONPath, but we should stick with one throughout the text.

(cabo:
Clearly, we are selecting nodes, which contain their values.)

Stefan's topics: Syntax

— Is a dot-member name allowed to start with a DIGIT? ➔ No.

(cabo:
related: what does `.1` applied to `[1, 2, 3]` mean?)

```
dot-selector     = "." dot-member-first *dot-member-char
dot-member-name-first =
                      ALPHA /
                      "_"   /            ; _
                      %x80-10FFFF        ; any non-ASCII Unicode character
dot-member-name-char =
                      DIGIT /
                      ALPHA /
                      "_"   /            ; _
                      %x80-10FFFF        ; any non-ASCII Unicode character

DIGIT            =  %x30-39              ; 0-9
ALPHA            =  %x41-5A / %x61-7A    ; A-Z / a-z
```

- Are unions allowed to contain wildcards (no, as all others become redundant then), descendant-selectors (no) and filter-selectors (to do in separate PR)?

(manage syntactical ambiguity, though!)

```
union-selector = "[" ws union-entry 1*(ws "," ws union-entry) ws "]"

union-entry    =  ( quoted-member-name /
                    element-index      /
                    slice-index
                  )
```

— Discuss the proposed lean backward compatible syntax `[?<expr>]` in contrast to original `[?(<expr>)]`.

`https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base/pull/98#discussion_r649696672`

The parentheses create a level of human-readability that I'm afraid will be lost by removing them.
— Greg

Or could the use of parentheses be a convention instead?
(But do consider the interoperability impact!)
➔ no change!

— Whitespace

Needs to be explicitly allowed (ABNF).

```
grpent = [occur S] [memberkey S] type
       / [occur S] groupname [genericarg]  ; preempted by above
       / [occur S] "(" S group S ")"
```

Where should it be allowed (and be insignificant)?
(#24)
multi-line queries?

➔ Keep issue open for future PR (be generous with whitespace)
(Add note: To do (#24))

Semantics

— Do we need to specify the order of evaluation of the descendant-selector?

➔ Yes. Do in separate PR.

Want to be deterministic for testing.
But we don't want to require it.
Define deterministic order, and encourage its use.

What is our stance on implicit conversions?
(Emerging consensus was: No implicit conversions.)
What about implicit conversion to Boolean ("truthy")?
➔ No.
But then there is the existence aspect.

`https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base/pull/98#discussion_r649624505`

- How much out there will be broken by being strict?
- How much does the cleanup give us?
  - vs., can small concessions have a big benefit?

Extent of functionality (already discussed above)

— Discuss proposal of `in-op` operator

(cabo:
Note that the proposed syntax doesn't say what the production
`container` is supposed to be.
Same for `regex`.
All this should be over in the expression language.)

— Should comparison with structured values be allowed?

```
* Comparisons are restricted to primitive values `number`, `string`, `true`, `false`, `null`.
  Comparisons with complex values will fail, i.e. no selection occurs.
```

(cabo:
This first requires defining literal and/or constructor syntax for structured values.)

An expression should be able to operate on any JSON literal. I see no reason why @.foo == [1, 2] should be disallowed.
— Greg

— What's the opinion about supporting functions?

```
calc_val      = func "(" [rel-path-val / json-path] ")"
```

https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base/pull/98#discussion_r649700312

Related to this is:
How to access a specific member in `{...,"key":5,...}`
via filter expression `[?@...]` ?

— extent of `rel-path-val`

$[?(@..foo contains 42)]

`https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base/pull/98#discussion_r649710822`

— filter selectors in unions?

`https://github.com/ietf-wg-jsonpath/draft-ietf-jsonpath-base/pull/98#discussion_r649694182`

➔ left to further PR