JSON Reduce Reference

Generated by Doxygen

20230422T093409Z

1 Overview

The JSON Reduce expression language provides a way to encode expressions which can be evaluated to JSON types, in JSON itself, in a relatively natural manner. It is similar to LISP S-expressions and the JSON script language designed by OpenWrt procd maintainers. However, it has the following differences from both:

- expressions have no side effects;
- there are no exceptions: failing expressions reduce to `null` instead;
- subexpressions can be and actually are evaluated in an unspecified order, depending on the availability of data to process and possibly on external policies as well;
- partially evaluated expressions can be passed along to hide details of the initial expression which the receiver need not, or must not, know about;
- expressions can be written in a way suitable for automated metaprocessing into e.g. a subset of natural language.
- it is by itself very simple; most useful features are expected to be supplied by applications.

The main use of JSON Reduce expressions is in evaluating DLI type description traits, constraints and access settings.

2 Reduction

A JSON Reduce expression can contain:

- expressions which are completely evaluated and cannot be reduced any further (values);
- expressions which should be reduced, which are further divided into:

  - expressions which can be reduced (the evaluator understands them and their arguments are sufficiently evaluated);
  - expressions which cannot currently be reduced.

During evaluation of a JSON Reduce expression, it is reduced to a simple value in a series of reduction steps. It may be impossible to reach a simple value (if no expressions can be reduced further, and the expression is still not a value). The expression remains a valid JSON Reduce expression on each step.

The order of reduction is only limited by nesting and requirements of the functions to evaluate. For example, if a logical "and" function detects that one of the arguments is `false`, it reduces to `false` itself, even without evaluating all the other arguments.

As a general rule, if a function receives an invalid set of operands, it reduces to `null`.

3 Syntax

3.1 Simple values

The following JSON types are values in JSON Reduce, and stand for themselves:

- strings (e.g. `"foo"`);
- numbers (e.g. `1.5`);
- constants `null`, `true` and `false`.

3.2 Function calls

In JSON Reduce, function calls are encoded using a JSON array, the first element of which specifies the function name, and the rest are the function arguments.

Not all functions require all of their arguments to be completely evaluated.

3.3 Compound values

JSON objects and arrays need to be quoted to be treated as values. The special function "quote" is used for that. For example, the JSON Reduce expression `["quote", {"a":"b"}]` cannot be reduced further, it is a value, and it has the value `{"a":"b"}`.

For consistency, simple values can be quoted too, e.g. `["quote","foo"]` has the value `"foo"`. The result of evaluation of `["quote"]` (without arguments) is the constant `null`.

4 Built-in functions

The JSON Reduce expression syntax does not require implementations to supply any functions except `"quote"`, but there are some standard general-purpose functions which should be implemented.

4.1 Comparison operators

Operators `"<"`, `">"`, `"<="`, `">="`, `"=="`, `"!="` have the expected meaning. They evaluate to either `true` or `false`. If there are less then 2 arguments, they evaluate to `true`. If there are more than two arguments `"a1"`,"a2","a3",... they have the following meaning:

- `"<"`, `">"`, `"<="`, `">="`, `"=="` : `a1 < a2 < a3 <` ..., etc.;
- `"!="` : `a1`, `a2`, `a3` are pairwise distinct (`a1 != a2`, `a1 != a3`, ..., `a2 != a3`, ...).

The general semantics is the triangular one as in the `"!="` case, it's just that its application is simplified for `"<"`, `">"`, `"<="`, `">="`, `"=="`.

Not all of the arguments need to be fully evaluated: functions may reduce to `false` if the condition is `false` for one of the already evaluated pairs.

`"=="` and `"!="` can be applied to JSON objects and arrays, checking for deep equality (as explained in e.g. JSON patch RFC).

4.2 Logic operators

Operators `"and"`, `"or"`, `"not"` have the expected meaning. However, the boolean value of common values differs from what is used in e.g. JavaScript. Only `null` and `false` have the logical value of `false`; all other values, in particular including `0`, the empty string `""`, the empty array `[]` and object `{}` are `true` in a boolean context.

Both `"and"` and `"or"` accept an arbitrary number of arguments, not all of which need to be fully evaluated. The first argument to evaluate to a `false` (for `"and"`) or `true` (for `"or"`) value makes the operator itself evaluate to `false` or `true`, respectively. If all arguments turn out to evaluate to `true` (for `"and"`) or `false` (for `"or"`) values, the whole expression evaluates to `true` (for `"and"`) or `false` (for `"or"`). During evaluation, it is possible for the evaluator to remove arguments which evaluate to `true` (for `"and"`) or `false` (for `"or"`). Note that the value of the operator is always either `true` or `false`, and it does not pass through any actual values of the arguments.

The `"not"` operator requires its only argument to be fully evaluated, and returns the negation of its logic value.

4.3 Arithmetic operators

Operators `"+"`, `"-"`, `"*"`, `"/"`, `"%"` (remainder) and `"^"` have the expected meaning; each requires numeric operands which must be completely evaluated. `"+"` and `"*"` support a variable number of arguments; others require exactly 2. If there are less then 2 arguments, `"+"` and `"*"` evaluate to the first argument, or to the neutral element (0 for `"+"`, 1 for `"*"`) if it's absent (no arguments).

The remainder operation `a % b` is defined as `a - floor(a / b) * b`, and works on fractional arguments as well.

A convenience function `"integer"` requires exactly one numeric argument and is reduced like `["integer",a] => ["==",["%",a,1],0]`. It checks whether its argument is a whole number.

The `"cat"` function is used for string concatenation. It supports a variable number of arguments. If there are less then 2 arguments, `"cat"` evaluates to the first argument, or to the neutral element (`""`) if it's absent (no arguments).

4.4 Container-related operators

The `"#"` function requires its operand to be a fully evaluated JSON array. It returns the number of elements in it.

The `"[]"` function is used to perform (possibly nested) indexing of compound types, like `a[2]["x"]`. It has the following behaviour:

- `["[]"]` evaluates to `null`;
- `["[]",a]` evaluates to `a`;
- `["[]",a,b,...]`, if `a` and `b` are fully evaluated, reduces to `["[]",a[b],...]`.

If at some point in the evaluation, the index is of an invalid type (not a number of a string), or the indexed value is not of the correct type (object for string keys and array for numeric keys), the whole expression evaluates to `null`.

The `"object"` function is used to construct JSON objects. Its arguments are expected to evaluate to two-element arrays of the form `[key,value]`. If non-string, non-null keys are found, the whole expression evaluates to `null`. Otherwise, when all arguments are fully evaluated, the resulting object is constructed from pairs which have non-null keys (pairs with `null` keys are silently ignored).

The `"array"` function is used to construct JSON arrays. It evaluates to an array composed of its arguments as soon as all of them are evaluated.

The `"#"` and `"[]"` functions may interact with `"object"` and `"array"` functions to perform shortcut evaluation.

4.5 String-related functions

The `"match"` function takes two fully evaluated string arguments, the pattern and the string to check. It evaluates to `true` if the string to check matches the pattern, interpreted as a POSIX extended regular expression, and to `false` otherwise.

4.6 Higher-order functions

Functions can be applied to arguments calculated elsewhere using the `"apply"` function. It takes the called function name as the first argument, and the array of arguments as the second argument, and is reduced like `["apply","fn",[a1,a2...]]` => `["fn",a1,a2...]`.

The second argument must be an array. Implementations may impose additional restrictions on `"apply"` arguments, e.g. that its first argument is a string literal with a limited set of values. If the requirements are not met, the expression evaluates to `null`.