Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 22, 2016

A. Newton
ARIN
P. Cordell
Codalogic

March 21, 2016

A Language for Rules Describing JSON Content
draft-newton-json-content-rules-06

Abstract

This document describes a language for specifying and testing the expected content of JSON structures found in JSON-using protocols, software, and processes.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 22, 2016.

1. Introduction

This document describes JSON Content Rules (JCR), a language for specifying and testing the interchange of data in JSON [RFC7159] format used by computer protocols and processes. The syntax of JCR is not JSON but is "JSON-like", possessing the conciseness and utility that has made JSON popular.

1.1. A First Example: Specifying Content

The following JSON data describes a JSON object with two members, "line-count" and "word-count", each containing an integer.

```
{ "line-count" : 3426, "word-count" : 27886 }
```

This is also JCR that describes a JSON object with a member named "line-count" that is an integer that is exactly 3426 and a member named "word-count" that is an integer that is exactly 27886.

For a protocol specification, it is probably more useful to specify that each member is any integer and not specific, exact integers:

```
{ "line-count" : integer, "word-count" : integer }
```

Since line counts and word counts should be either zero or a positive integer, the specification may be further narrowed:

```
{ "line-count" : 0.. , "word-count" : 0.. }
```

1.2. A Second Example: Testing Content

Building on the first example, this second example describes the same object but with the addition of another member, "file-name".

```
{
  "file-name"  : "rfc7159.txt",
  "line-count" : 3426,
  "word-count" : 27886
}
```

The following JCR describes objects like it.

```
{
  "file-name"  : string,
  "line-count" : 0..,
  "word-count" : 0..
}
```

For the purposes of writing a protocol specification, JCR may be broken down into named rules to reduce complexity and to enable re-use. The following example takes the JCR from above and rewrites the members as named rules.

```
{
  fn,
  lc,
  wc
}

fn "file-name"  : string
lc "line-count" : 0..
wc "word-count" : 0..
```

With each member specified as a named rule, software testers can override them locally for specific test cases. In the following example, the named rules are locally overridden for the test case where the file name is "rfc4627.txt".

```
fn "file-name"  : "rfc4627.txt"
lc "line-count" : 2102
wc "word-count" : 16714
```

In this example, the protocol specification describes the JSON object in general and an implementation overrides the rules for testing specific cases.

2. Overview of the Language

JCR is composed of rules (as the name suggests). A collection of rules that is processed together is a ruleset. There are five types of rules: value rules, member rules, array rules, object rules, and group rules. The first four types describe corresponding aspects of JSON, respectively.

Except for an optional root rule, each rule has two components, a rule name and a rule definition:

```
<rule name> <rule definition>
```

Rule definitions may in turn contain child rule definitions or reference other rules by their rule name.

This is an example of a value rule:

```
v1 : 0..3
```

It specifies a rule named "v1" that has a definition of ": 0..3" (value rule definitions begin with a ':' character). This defines values of type "v1" to be integers in the range 0 to 3 (minimum value of 0, maximum value of 3). Value rules can define the limits of JSON values, such as stating that numbers must fall into a certain range or that strings must be formatted according to certain patterns or standards (i.e. URIs, IP addresses, etc...).

Member rules specify JSON object members. The following example member rule states that the rule's name is 'm1' with a value defined by the rule named 'v1':

```
m1 "m1name" v1
```

Since rule names are substituted by rule definitions, this member rule can also be written as follows (to define a member rule named m1 for JSON member named "m1name" that has a value that is an integer between 0 and 3):

```
m1 "m1name" : 0..3
```

Object rules are composed of member rules, since JSON objects are composed of members. Object rules can specify members that are mandatory, optional, and even choices between members. In this example, the rule 'o1' defines an object that must contain a member as defined by member rule 'm1' and optionally a member defined by the rule 'm2':

```
o1 { m1, ?m2 }
```

Array rules are composed of value, object, and other array rules. Like object rules, array rules can specify the cardinality of the contents of an array. The following array rule defines an array that must contain value rule 'v1' and zero or more objects as defined by rule 'o1':

```
a1 [ v1, *o1 ]
```

Finally, group rules designate a collection of rules.

Putting it all together, Figure 2 describes the JSON in Figure 1.

Example JSON shamelessly lifted from RFC 4627

```
{
  "Image": {
      "Width":  800,
      "Height": 600,
      "Title":  "View from 15th Floor",
      "Thumbnail": {
          "Url":    "http://www.example.com/image/481989943",
          "Height": 125,
          "Width":  "100"
      },
      "IDs": [116, 943, 234, 38793]
   }
}
```

Figure 1

Rules describing Figure 1

```
{ image }

image "Image" {
    width,
    height,
    "Title" : string,
    thumbnail,
    "IDs" [ *: integer ]
}

width "Width" width_v
height "Height" height_v

width_v : 0..1280
height_v : 0..1024

thumbnail "Thumbnail" {
    width, height, "Url" : uri
}
```

Figure 2

The rules from Figure 2 can be written more compactly (see Figure 3).

Compact rules describing Figure 1

```
{
    "Image" {
        width,
        height,
        "Title" :string,
        "Thumbnail" {
            width, height, "Url" :uri
        },
        "IDs" [ *:integer ]
    }
}

width "Width" : 0..1280
height "Height" : 0..1024
```

Figure 3

3. Lines and Comments

There is no statement terminator and therefore no need for a line continuation syntax. Rules may be defined across line boundaries. Blank lines are allowed.

Comments are very similar to comments in ABNF [RFC4234]. They start with a semi-colon (';') and continue to the end of the line or another semi-colon.

4. Rules

Rules are composed of two parts, a rule name and a rule definition:

```
<rule name> <rule definition>
```

Rule names allow a rule to be identified by a name. A rule definition describes the constraints upon which the content is to be assessed. Rule definitions can use rule names to refer to other rules.

4.1. Rule Names

Rule names must start with an alphabetic character (a-z,A-Z) and must contain only alphabetic characters, numeric characters, the hyphen character ('-') and the underscore character ('_').

Rule names are case sensitive. Rule names identifying rule definitions must be unique within a ruleset.

4.2. Rule Definitions

The syntax of each type of rule definition varies depending on the type:

```
: string
; value rules start with a colon

"member_name" target_rule_name
; member rules start by defining the member name

{ mem1, mem2 }
; object rules start and end with "curly braces", like JSON objects

[ item1, item2 ]
; array rules start and end with square brackets, like JSON arrays

( rule1, rule2 )
; group rules start and end with parenthesis
```

A rule definition may embed other rule definitions, either explicitly or by referencing a rule name that identifies a rule definition.

4.3. Annotations

Rule definitions may start with zero or more annotations. Each annotation begins with the character sequence "@{" and ends with "}". The following is an example of a rule definition with the root annotation (explained in the next section):

```
@{root} [ nuts, bolts ]
```

This specification defines the annotations "root", "reject", and "unordered", but other annotations may be defined.

4.4. Starting Points and Root Rules

Careful readers will have noticed that although rules have been defined as having rule names and rule definitions, examples from the introduction have one rule without a rule name. Within each ruleset, a name on the first rule is optional. When the first rule is defined without a name, it is considered a root rule.

Root rules are a starting point for the evaluation of JSON against a ruleset. Or in other words, a root rule is the first rule processed.

Rules may also be declared a root rule with the @{root} annotation. A ruleset may have more than one root rule, in which case the root

rule to use for validating JSON should be explicitly specified locally.

4.5. Value Rules

Value rules define content for JSON values. JSON allows values to be objects, arrays, numbers, booleans, strings, and null. Arrays and objects are handled by the array and object rules, and the value rules define the rest.

4.5.1. Numbers, Booleans and Null

The rules for booleans and null are the simplest and take the following forms:

```
rule_name : true

rule_name : false

rule_name : boolean

rule_name : null
```

Rules for numbers can specify the number be either an integer or floating point number:

```
rule_name : integer

rule_name : float
```

Numbers may also be specified as an absolute value or a range of possible values, where a range may be specified using a minimum, maximum, or both:

```
rule_name : n

rule_name : n..m

rule_name : ..m

rule_name : n..

rule_name : n.f..m.f

rule_name : ..m.f

rule_name : n.f..
```

When specifying a minimum and a maximum, both must either be an integer or a floating point number. Thus to specify a floating point number between zero and ten a definition of the following form is used:

```
: 0.0..10.0
```

4.5.2. Strings

String values may be specified generically as:

```
rule_name : string
```

However, the content of strings can be narrowed in the following ways:

A quoted string: A rule can specify that the value must be a specific string:

```
rule_name : "a constant string"
```

Regular Expression: A rule can state that a string must match a regular expression by giving the regular expression:

```
rule_name : /regex/
```

URIs and URI templates: A rule can state that a string must be a URI [RFC3986]:

```
rule_name : uri
```

URIs may be further scoped to a specific URI pattern by using a URI template [RFC6570]:

```
rule_name : uri..http://{stuff}
```

```
rule_name : uri..http://{authority}/{thing1}?q={thing2}
```

When using URI templates, the variable names are ignored for pattern matching, but they should be provided for construction of a valid URI template. Providing the variable names also aids in the description of what is to be matched.

IP Addresses: Narrowing the content of strings down to IP addresses can be done with either the 'ip4' (see [RFC1166]) or 'ip6' (see [RFC5952]) literals:

```
rule_name : ip4
```

```
rule_name : ip6
```

Domain Names: Fully qualified A-label and U-label domain names can be specified with the 'fqdn' and 'idn' literals:

```
rule_name : fqdn

rule_name : idn
```

Dates and Times: Dates and times are specified using the ABNF rules from RFC 3339 [RFC3339] as literals:

```
rule_name : date-time

rule_name : full-date

rule_name : full-time
```

Email Addresses: A string can be scoped to the syntax of email addresses using the literal 'email':

```
rule_name : email
```

Email addresses must conform to the syntax of RFC 5322 [RFC5322].

Phone Numbers: Strings conforming to E.123 phone number format can be specified as follows:

```
rule_name : phone
```

Base 64: Strings containing base 64 data, as described by RFC 4648 [RFC4648], can be specified as follows:

```
rule_name : base64
```

4.5.3. Any Value

It is possible to specify that a value can be of any type allowable by JSON using the 'any' value rule. This is done with the 'any' literal in a value rule:

```
rule_name : any
```

However, unlike other value rules which define primitive data types, this rule defines a value of any kind, either primitive (null, boolean, number, and string), object, or array.

4.6. Member Rules

Member rules define members of JSON objects. Member rules follow the format:

```
rule_name member_name type
```

where rule_name is the name of the rule being defined, member_name is the name of the JSON object member, and type is a value rule, array rule, or object rule or a reference to a value rule, array rule, or object rule specifying the allowable content of the JSON object member.

Member names may be specified either explicitly as a quoted string:

```
some_member_rule "some_member_name" some_member_target
```

or a family of member names may be specified as a regular expression:

```
some_member_rule /some\.[a-z]+\.names/ some_member_target
```

Member rules may also be written in this form:

```
rule_name "member_rule" target_rule_definition
```

The following are examples:

```
location_uri "locationURI" : uri
```

```
iface_mappings /eth[0-9]/ :ip4
```

Member rules cannot be used as a root rule.

4.7. Object Rules

Object rules define the allowable members of a JSON object, and their rule definitions contain the member rules of the object. They take the following form:

```
rule_name { member_rule_1, member_rule_2 }
```

The following rule example defines an object composed of two member rules:

```
response { location_uri, status_code }
```

Given that where a rule name is found a rule definition of an appropriate type may be used, the above example might also be written as:

```
response { "locationUri" : uri, "statusCode" : integer }
```

Rules given in the rule definition of an object rule do not imply order. Given the example object rule above both

```
{ "locationUri" : "http://example.com", "statusCode" : 200 }
```

and

```
{ "statusCode" : 200, "locationUri" : "http://example.com" }
```

are JSON objects that match the rule.

Each member rule of an object rule is evaluated in the order in which they appear in the object rule. Thus where there is potential conflict between rule names defined using regular expressions, the rules with the most constrained name should be defined first. Otherwise, for example, a rule definition of:

```
{ /p\d+/ : int, "p0" : string }
```

would fail to match the JSON object:

```
{ "p1" : 12, "p0" : "Fred" }
```

because the "p0" member name would match the regular expression despite the presence of the subsequently defined "p0" member rule.

4.8. Array Rules

Array rules define the allowable content of JSON arrays. Their rule definitions are composed of the other rule types with the exception of member rules and have the following form:

```
rulename [ rule_1, rule_2 ]
```

The following example defines an array where the first element is defined by the width_value rule and the second element is defined by the height_value rule:

```
size [ width_value, height_value ]
```

By default, unlike object rules, order is implied by the array rule definition. That is, the first rule referenced or defined within an

array rule specifies that the first element of the array will match that rule, the second rule given with the array rule specifies that the second element of the array will match that rule, and so on.

Take for example the following array rule definition:

```
person [ : string, : integer ]
```

This JSON array matches the above rule:

```
[ "Bob Smurd", 24 ]
```

while this one does not:

```
[ 24, "Bob Smurd" ]
```

Finally, if an array has more elements than can be matched from the array rule, the array does not match the array rule. Or stated differently, an array with unmatched elements does not validate. Using the example array rule above, the following array does not match because the last element of the array does not match any rule contained in the array rule:

```
[ "Bob Smurd", 24, "http://example.com/bob-smurd" ]
```

4.8.1. Unordered Array Rules

Array rules can be made to behave in a similar fashion to object rules with regard to the order of matching with the @{unordered} annotation:

```
person @{unordered} [ :string, :integer ]
```

This rule matches both of theses JSON arrays.

```
[ "Bob Smurd", 24 ]
```

```
[ 24, "Bob Smurd" ]
```

Like ordered array rules, the rules contained in an unordered array rule are evaluated in the order they are specified. The difference is that they need not match an element of the array in the same position as given in the array rule.

Like ordered array rules, unordered array rules also require that all elements of the array be matched by a subordinate rule. If the array has more elements than can be matched, the array rule does not match the array.

4.9. Group Rules

Unlike the other types of rules, group rules have no direct tie with JSON syntax. Group rules simply group together other rules. They take the form:

```
rule_name ( target_rule_1, target_rule_2 )
```

Group rule definitions and any nesting of group rule definitions, must conform to the allowable set of rules of the rule containing them. A group rule referenced inside of an array rule may not contain a member rule since member rules are not allowed in array rules directly. Likewise, a group rule referenced inside an object rule must only contain member rules.

The following is an example of a group rule:

```
the_bradys [ parents, children ]
children ( :"Greg", :"Marsha", :"Bobby", :"Jan" )
parents ( :"Mike", :"Carol" )
```

Like the subordinate rules of array and object rules, the subordinate rules of a group rule are evaluated in the order they appear.

4.10. Ordered and Unordered Groups in Arrays

Section 4.8.1 specifies that arrays can be evaluated by the order of the items in the array or can be evaluated without order. Section 4.9 specifies that arrays may have group rules as subordinates.

The evaluation of a group rule inside an array rule inherits the ordering property of the array rule. If the array rule is unordered, then the items of the group rule are also considered to be unordered. And if the array rule is ordered, then the items of the group rule are also considered to be ordered.

4.11. Sequence and Choice Combinations in Array, Object, and Group Rules

Combinations of subordinate rules in array, object, and group rules can be specified as either a sequence ("and") or a choice ("or"). A sequence is a rule followed by the comma character (',') followed by another rule.

```
[ this, that ]
```

A choice is a rule followed by a pipe character ('|') followed by another rule.

```
[ this | that ]
```

Sequence and choice combinations cannot be mixed, and group rules must be used to explicitly declare precedence between a sequence and a choice. Therefore, the following is illegal:

```
[ this, that | the_other ]
```

The example above should be expressed as:

```
[ this, ( that | the_other ) ]
```

4.12. Repetition in Array, Object, and Group Rules

Evaluation of subordinate rules in array, object, and group rules may be preceded by a repetition expression denoting how many times the subordinate rule should be evaluated.

Repetition is expressed as a minimum number of repetitions and a maximum number of repetitions. When no repetition expression is present, both the minimum and maximum are 1.

A minimum and maximum can be expressed by giving the minimum followed by an asterisk ('*') character followed by the maximum: min*max.

```
[ 1*13 name_servers ] ; 1 to 13 name servers
```

If the minimum is not given, it is assumed to be zero.

```
{ *99 /eth.*/ mac_addr }; 0 to 99 ethernet addresses
```

If the maximum is not given, it is assumed to be infinity.

```
[ 2* octets ] ; two or more bytes
```

If neither the minimum nor the maximum are given with the asterisk, this denotes "zero or more".

```
error_set ( * error ) ; zero or more errors
```

Repetition may also be expressed with a question mark character ('?') or a plus character ('+'). '?' is equivalent to '0*1'.

```
{ name, ?age } ; age is optional
```

'+' is equivalent to '1*'

```
   [ + status ] ; 1 or more status values
```

4.13. Rejecting Rules

The evaluation of a rule can be changed with the @{reject} annotation. With this annotation, a rule that would otherwise match does not, and a rule that would not have matched does.

```
    not_two @{reject} : 2
    ; match anything that isn't the integer 2

    @{reject} @{unordered} [ :"fail", *:string ]
    ; error if one of the status values is "fail"
```

4.14. Repetitions, Annotations, and Target Rules

With regard to syntax, repetition expressions are part of the syntax of array, object, and group rules with respect to the embedding of subordinate rules, whereas annotations are a component of every type of rule definition. Every type of rule definition may begin with a series of annotations.

The significance is the placement of repetition expressions with respect to annotations: repetition expressions precede annotations.

The following is correct:

```
   [ * @{unordered} [ foo ] ]
```

The following is not:

```
   [ @{unordered} * [ foo ] ]
```

5. Directives

Directives modify the processing of a ruleset. There are two forms of the directive, the single line directive and the multi-line directive.

Single line directives appear on their own line in a ruleset, begin with a hash character ('#') and are terminated by the end of the line. They take the following form:

```
   # directive_name optional_directive_parameters
```

Directives may have other qualifiers after the directive name.

Multi-line directives also appear on their own lines, but may span multiple lines. The being with the character sequence "#{" and ends with "}". The take the following form:

```
#{ directive_name
     directive_parameter_1 directive_paramter_2
     directive_parameter_3
     ...
}
```

This specification defines the directives "jcr-version", "ruleset-id", and "import", but other directives may be defined.

5.1. jcr-version

This directive declares that the ruleset complies with a specific version of this standard. The version is expressed as a major integer followed by a period followed by a minor integer.

```
# jcr-version 0.6
```

The major.minor number signifying compliance with this document is "0.6". Upon publication of this specification as an IETF proposed standard, it will be "1.0".

```
# jcr-version 1.0
```

Ruleset authors are advised to place this directive as the first line of a ruleset.

5.2. ruleset-id

This directive identifies a ruleset to rule processors. It takes the form:

```
# ruleset-id identifier
```

An identifier can be a URL (e.g. http://example.com/foo), an inverted domain name (e.g. com.example.foo) or any other form that conforms to the JCR ABNF syntax that a ruleset author deems appropriate. To a JCR processor the identifier is treated as an opague, case-sensitive string.

5.3. import

The import directive specifies that another ruleset is to have its rules evaluated in addition to the ruleset where the directive appears.

This directive has the following form:

```
# import identifier as alias
```

The following is an example:

```
# import http://example.com/rfc9999 as rfc9999
```

The rule names of the ruleset to be imported may be referenced by prepending the alias followed by a period character ('.') followed by the rule name (i.e. "alias.name"). To continue the example above, if the ruleset at http://example.com/rfc9999 were to have a rule named 'encoding', rules in the ruleset importing it can refer to that rule as 'rfc9999.encoding'.

6. Tips and Tricks

6.1. Any Member with Any Value

Because member names may be specified with regular expressions, it is possible to construct a member rule that matches any member name:

```
rule_name /.*/ target_rule_name
```

As an example, the following defines an object member with any name that has a value that is a string:

```
user_data /.*/ : string
```

Constructing an object member of any name with any type would therefore take the form:

```
rule_name /.*/ : any
```

6.2. Restricting Objects

By default, members of objects which do not match a rule are ignored. The reason for this validation model is due to the nature of the typical access model to JSON objects in many programming languages, where members of the object are obtained by referencing the member name. Therefore extra members may exist without harm.

However, some specifications may need to restrict the members of a JSON object to a known set. To construct an object rule specifying that no extra members are expected, the @{reject} annotation may be used with a regular expression as the last subordinate rule of the object rule.

```
{ member1, member2, + @{reject} /.*/ : any }
```

This works because subordinate rules are evaluated in the order they appear in the object rule, and the last rule accepts any member with any type but fails to validate if one ore more of those rules are found due to the @{reject} annotation.

6.3. Unrestricting Arrays

Unlike object validation, array rules will not validate items of an array that do not match a subordinate rule of the array rule. This processing model is due to the nature of the typical access pattern of JSON arrays in many programming languages, which is to iterate over the array. Processes iterating over an array would need to take special steps for extra items of the array that are not specified, especially if the items were of a different type than those that are expected.

Like object rules, the subordinate rules of an array rule are evaluated in the order they appear. To allow an array to contain any value after guaranteeing that it contains the necessary items, the last subordinate rule of the array rule should accept any item:

```
[ item1, item2, * :any ]
```

6.4. Groups of Values

In addition to specific primitive data types, value rules may contain a value choice rule. The value choice rule, and any subordinate rule within it, must evaluate to a single primitive data type.

The following is an example of a value choice rule embedded in a value rule:

```
address : ( :ip4 | :ip6 )
```

6.5. Groups in Arrays

Groups may also be a subordinate rule of array rules:

```
[ ( :ip4 | :ip6 ), :integer ]
```

Unlike value rules, subordinate group rules in array rules may have sequence combinations and contain any rule type with the exception of member rules.

```
[ ( first_name, ? middle_name, last_name ), age ]
```

Of course, the above is better written as:

```
[ name, age ]

name ( first_name, ? middle_name, last_name )
```

6.6. Groups in Objects

Groups may also be a subordinate rule of object rules:

```
{ ( title, date, author ), + paragraph }
```

Subordinate group rules in object rules may have sequence combinations but must only contain member rules.

```
 { front_matter, + paragraph }
 front_matter ( title, date, author )
 title "title" :string
 date "date" : full-date
 author "author" [ *:string ]
 paragraph /p[0-9]*/ :string
```

6.7. Group Rules as Macros

The syntax for group rules accommodates one ore more subordinate rules and a repetition expression for each. Other than grouping multiple rules, a group rule can be used as a macro definition for a single rule.

```
paragraphs ( + /p[0-9]*/ : string )
```

6.8. Comment Separated Rules

Rules may be placed on the same line, but because they have no termination syntax this style of writing rules can be confusing to some readers:

```
first_name "first name" :string last_name "last name" :string
```

An empty comment can serve as a visual cue to denote the separation of the two rules:

```
first_name "first name" :string ;; last_name "last name" :string
```

6.9. Object Mixins

Group rules can be used to create object mixins, a pattern for writing data models similar in style to object derivation in some programming languages. In the example in Figure 4, both obj1 and obj2 have a members "foo" and "fob" with obj1 having the additional member "bar" and obj2 having the additional member "baz".

```
mixin_group ( "foo" : integer, "fob" : uri )

obj1 { mixin_group, "bar" : string }

obj2 { mixin_group, "baz" : string }
```

Figure 4

6.10. Subordinate Rule Dependencies

In object and array rules, there may be situations in which it is necessary to condition the existence of a subordinate rule on the existence of a sibling subordinate rule. In other words, example_rule_two should only be evaluated if example_rule_one evaluates positively. Or put another way, a member of an object or an item of an array may be present only on the condition that another member of item is present.

In the following example, the referrer_uri member can only be present if the location_uri member is present.

```
response { ?( location_uri, ?referrer_uri ) }
```

6.11. Multiple Root Styles

As stated in Section 4.4, the first rule in a ruleset is a root rule when it is unnamed. Group rules can be used as the root rule, such as in the following example:

```
( { "foo" : string } | { "bar" : string } )
```

This is the equivalent of the following:

```
foo @{root} { "foo" : string }
bar @{root} { "bar" : string }
```

Either style is valid. However, explicitly naming root rules has the advantage of explicitly validating a JSON message against a specific rule.

6.12. JSON-like Object and Array Definitions

JCR allows an optional colon character (":") to precede object and array rule definitions to give these definitions more of a JSON-like appearance. Consider the following example.

```
    {
        "foo" {
           "fuzz" : string
         },
         "bar" [
            "baz"
         ]
     }
```

To appear more JSON-like, this may also be given as follows:

```
    {
        "foo" : {
           "fuzz" : string
         },
         "bar" : [
            "baz"
         ]
     }
```

7. ABNF Syntax

The following ABNF describes the syntax for JSON Content Rules.

```
jcr              = *( sp-cmt / directive ) [ root-rule ]
                   *( sp-cmt / directive / rule )

sp-cmt           = spaces / comment
spaces           = 1*( WSP / CR / LF )
comment          = ";" *( "\;" / comment-char ) comment-end-char
comment-char     = HTAB / %x20-3A / %x3C-10FFFF
                   ; Any char other than ";" / CR / LF
comment-end-char = CR / LF / ";"

directive        = "#" (one-line-directive / multi-line-directive)
one-line-directive = [ spaces ]
                   (directive-def / one-line-tbd-directive-d) *WSP eol
multi-line-directive = "{" *sp-cmt
                   (directive-def / multi-line-tbd-directive-d) *sp-cmt "}"
directive-def    = jcr-version-d / ruleset-id-d / import-d
jcr-version-d    = jcr-version-kw spaces major-version "." minor-version
major-version    = p-integer
```

```
minor-version    = p-integer
ruleset-id-d     = ruleset-id-kw spaces ruleset-id
import-d         = import-kw spaces ruleset-id
                   [ spaces as-kw spaces ruleset-id-alias ]
ruleset-id       = ALPHA *not-space
not-space        = %x21-10FFFF
ruleset-id-alias = name
one-line-tbd-directive-d = directive-name [ WSP one-line-directive-parameters ]
directive-name   = name
one-line-directive-parameters = *not-eol
not-eol          = HTAB / %x20-10FFFF
eol              = CR / LF
multi-line-tbd-directive-d = directive-name
                   [ spaces multi-line-directive-parameters ]
multi-line-directive-parameters = multi-line-parameters
multi-line-parameters = *(comment / q-string / regex /
                   not-multi-line-special)
not-multi-line-special = spaces / %x21 / %x23-2E / %x30-3A / %x3C-7C /
                   %x7E-10FFFF ; not ", /, ; or }

root-rule        = value-rule / group-rule

rule             = rule-name *sp-cmt rule-def

rule-name        = name
target-rule-name = annotations [ ruleset-id-alias "." ] rule-name
name             = ALPHA *( ALPHA / DIGIT / "-" / "-" )

rule-def         = type-rule / member-rule / group-rule
type-rule        = value-rule / type-choice-rule / target-rule-name
value-rule       = primitive-rule / array-rule / object-rule
member-rule      = annotations
                   member-name-spec *sp-cmt type-rule
member-name-spec = regex / q-string
type-choice-rule = ":" *sp-cmt type-choice
type-choice      = annotations "(" type-choice-items
                   *( choice-combiner type-choice-items ) ")"
type-choice-items = *sp-cmt ( type-choice / type-rule ) *sp-cmt

annotations      = *( "@{" *sp-cmt annotation-set *sp-cmt "}" *sp-cmt )
annotation-set   = reject-annotation / unordered-annotation /
                   root-annotation / tbd-annotation
reject-annotation = reject-kw
unordered-annotation = unordered-kw
root-annotation  = root-kw
tbd-annotation   = annotation-name [ spaces annotation-parameters ]
annotation-name  = name
annotation-parameters = multi-line-parameters
```

```
primitive-rule   = annotations ":" *sp-cmt primimitive-def
primimitive-def  = null-type / boolean-type / true-value / false-value /
                   string-type / string-range / string-value /
                   float-type / float-range / float-value /
                   integer-type / integer-range / integer-value /
                   ip4-type / ip6-type / fqdn-type / idn-type /
                   uri-range / uri-type / phone-type / email-type /
                   full-date-type / full-time-type / date-time-type /
                   base64-type / any
null-type        = null-kw
boolean-type     = boolean-kw
true-value       = true-kw
false-value      = false-kw
string-type      = string-kw
string-value     = q-string
string-range     = regex
float-type       = float-kw
float-range      = float-min ".." [ float-max ] / ".." float-max
float-min        = float
float-max        = float
float-value      = float
integer-type     = integer-kw
integer-range    = integer-min ".." [ integer-max ] / ".." integer-max
integer-min      = integer
integer-max      = integer
integer-value    = integer
ip4-type         = ip4-kw
ip6-type         = ip6-kw
fqdn-type        = fqdn-kw
idn-type         = idn-kw
uri-range        = uri-dotdot-kw uri-template
uri-type         = uri-kw
phone-type       = phone-kw
email-type       = email-kw
full-date-type   = full-date-kw
full-time-type   = full-time-kw
date-time-type   = date-time-kw
base64-type      = base64-kw
any              = any-kw

object-rule      = annotations [ ":" *sp-cmt ] "{" *sp-cmt [ object-items *sp-
cmt ] "}"
object-items     = object-item (*( sequence-combiner object-item ) /
                   *( choice-combiner object-item ) )
object-item      = [ repetition *sp-cmt ] object-item-types
object-item-types = member-rule / target-rule-name / object-group
object-group     = "(" *sp-cmt [ object-items *sp-cmt ] ")"

array-rule       = annotations [ ":" *sp-cmt ] "[" *sp-cmt [ array-items *sp-
cmt ] "]"
```

```
array-items      = array-item (*( sequence-combiner array-item ) /
                   *( choice-combiner array-item ) )
array-item       = [ repetition ] *sp-cmt array-item-types
array-item-types = type-rule / array-group
array-group      = "(" *sp-cmt [ array-items *sp-cmt ] ")"

group-rule       = annotations "(" *sp-cmt [ group-items *sp-cmt ] ")"
group-items      = group-item (*( sequence-combiner group-item ) /
                   *( choice-combiner group-item ) )
group-item       = [ repetition ] *sp-cmt group-item-types
group-item-types = type-rule / member-rule / group-group
group-group      = group-rule

sequence-combiner = *sp-cmt "," *sp-cmt
choice-combiner  = *sp-cmt "|" *sp-cmt

repetition       = optional / one-or-more / min-max-repetition /
                   min-repetition / max-repetition /
                   zero-or-more / specific-repetition
optional         = "?"
one-or-more      = "+"
zero-or-more     = "*"
min-max-repetition = min-repeat *sp-cmt "*" *sp-cmt max-repeat
min-repetition   = min-repeat *sp-cmt "*"
max-repetition   = "*" *sp-cmt max-repeat
min-repeat       = p-integer
max-repeat       = p-integer
specific-repetition = p-integer

integer          = ["-"] 1*DIGIT
p-integer        = 1*DIGIT

float            = [ minus ] int frac [ exp ]
                   ; From RFC 7159 except 'frac' required
minus            = %x2D                       ; -
plus             = %x2B                       ; +
int              = zero / ( digit1-9 *DIGIT )
digit1-9         = %x31-39                    ; 1-9
frac             = decimal-point 1*DIGIT
decimal-point    = %x2E                       ; .
exp              = e [ minus / plus ] 1*DIGIT
e                = %x65 / %x45                ; e E
zero             = %x30                       ; 0

q-string         = quotation-mark *char quotation-mark
                   ; From RFC 7159
char             = unescaped /
                   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
escape          = %x5C              ; \
quotation-mark  = %x22      ; "
unescaped       = %x20-21 / %x23-5B / %x5D-10FFFF

regex           = "/" *( escape "/" / not-slash ) "/" [ regex-modifiers ]
not-slash       = HTAB / CR / LF / %x20-2E / %x30-10FFFF
                  ; Any char except "/"
regex-modifiers = *( "i" / "s" / "x" )

uri-template    = 1*ALPHA ":" 1*not-space

;; Keywords
any-kw          = %x61.6E.79                    ; "any"
as-kw           = %x61.73                       ; "as"
base64-kw       = %x62.61.73.65.36.34           ; "base64"
boolean-kw      = %x62.6F.6F.6C.65.61.6E        ; "boolean"
date-time-kw    = %x64.61.74.65.2D.74.69.6D.65  ; "date-time"
email-kw        = %x65.6D.61.69.6C              ; "email"
false-kw        = %x66.61.6C.73.65              ; "false"
float-kw        = %x66.6C.6F.61.74              ; "float"
fqdn-kw         = %x66.71.64.6E                 ; "fqdn"
full-date-kw    = %x66.75.6C.6C.2D.64.61.74.65  ; "full-date"
full-time-kw    = %x66.75.6C.6C.2D.74.69.6D.65  ; "full-time"
idn-kw          = %x69.64.6E                    ; "idn"
import-kw       = %x69.6D.70.6F.72.74           ; "import"
integer-kw      = %x69.6E.74.65.67.65.72        ; "integer"
ip4-kw          = %x69.70.34                    ; "ip4"
ip6-kw          = %x69.70.36                    ; "ip6"
jcr-version-kw  = %x6A.63.72.2D.76.65.72.73.69.6F.6E ; "jcr-version"
null-kw         = %x6E.75.6C.6C                 ; "null"
phone-kw        = %x70.68.6F.6E.65              ; "phone"
reject-kw       = %x72.65.6A.65.63.74           ; "reject"
root-kw         = %x72.6F.6F.74                 ; "root"
ruleset-id-kw   = %x72.75.6C.65.73.65.74.2D.69.64 ; "ruleset-id"
string-kw       = %x73.74.72.69.6E.67           ; "string"
true-kw         = %x74.72.75.65                 ; "true"
unordered-kw    = %x75.6E.6F.72.64.65.72.65.64  ; "unordered"
uri-dotdot-kw   = %x75.72.69.2E.2E              ; "uri.."
uri-kw          = %x75.72.69                    ; "uri"
```

```
;; Referenced RFC 5234 Core Rules
ALPHA           = %x41-5A / %x61-7A   ; A-Z / a-z
CR              = %x0D         ; carriage return
DIGIT           = %x30-39      ; 0-9
HEXDIG          = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
HTAB            = %x09         ; horizontal tab
LF              = %x0A         ; linefeed
SP              = %x20         ; space
WSP             = SP / HTAB    ; white space
```

JSON Content Rules ABNF

8. Acknowledgements

Andrew Biggs and Paul Jones provided feedback and suggestions which led to many changes in the syntax.

9. References

9.1. Normative References

[RFC1166] Kirkpatrick, S., Stahl, M., and M. Recker, "Internet numbers", RFC 1166, DOI 10.17487/RFC1166, July 1990, <http://www.rfc-editor.org/info/rfc1166>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <http://www.rfc-editor.org/info/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC4234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 4234, DOI 10.17487/RFC4234, October 2005, <http://www.rfc-editor.org/info/rfc4234>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <http://www.rfc-editor.org/info/rfc5322>.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <http://www.rfc-editor.org/info/rfc5952>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <http://www.rfc-editor.org/info/rfc6570>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

9.2. Infomative References

[I-D.cordell-jcr-co-constraints]
Cordell, P. and A. Newton, "Co-Constraints for JSON Content Rules", draft-cordell-jcr-co-constraints-00 (work in progress), March 2016.

[ARIN_JCR_VALIDATOR]
American Registry for Internet Numbers, "JSON Content Rules Validator (Work In Progress)", <https://github.com/arineng/jcrvalidator>.

[CODALOGIC_JCR_VALIDATOR]
Codalogic, "cl-jcr-parser (Work In Progress)", <https://github.com/codalogic/cl-jcr-parser>.

Appendix A. Co-Constraints

This specification defines a small set of annotations and directives for JCR, yet the syntax is extensible allowing for other annotations and directives. [I-D.cordell-jcr-co-constraints] ("Co-Constraints for JCR") defines further annotations and directives which define more detailed constraints on JSON messages, including co-constraints (constraining parts of JSON message based on another part of a JSON message).

Appendix B. Testing Against JSON Content Rules

One aspect of JCR that differentiates it from other format schema languages are the mechanisms helpful to developers for taking a formal specification, such as that found in an RFC, and evolving it into unit tests, which are essential to producing quality protocol implementations.

B.1. Locally Overriding Rules

As mentioned in the introduction, one tool for testing would be the ability to locally override named rules. As an example, consider the following rule which defines an array of strings.

```
statuses [ * :string ]
```

Consider the specification where this rule is found does not define the values but references an IANA registry for extensibility purposes.

If a software developer desired to test a specific situation in which the array must at least contain the status "accepted", the rules from the specification could be used and the statuses rule could be explicitly overridden locally as:

```
statuses @{unordered} [ :"accepted", * :string ]
```

Alternatively, the developer may need to ensure that the status "denied" should not be present in the array:

```
statuses @{unordered} [ ? @{reject} :"denied", * :string ]
```

B.2. Rule Callbacks

In many testing scenarios, the evaluation of rules may become more complex than that which can be expressed in JCR, sometimes involving variables and interdependencies which can only be expressed in a programming language.

A JCR processor may provide a mechanism for the execution of local functions or methods based on the name of a rule being evaluated. Such a mechanism could pass to the function the data to be evaluated, and that function could return to the processor the result of evaluating the data in the function.

Appendix C. Combining Multiple Rulesets (Experimental)

```
This section is experimental and subject to further development.
```

Many work items within the IETF are defined by a core specification which is later enhanced by extension specifications. JCR supports this pattern of working by using the (@augments) annotation.

The parameters of the @{augments} annotation are a list of one or more target-rule-names that identify rules to be augmented. The augmentation process consists of logically adding a reference to the

rule name of the rule that contains the @{augments} annotation into each of the rules identified by the target-rule-names in the @{augments} annotation.

As an example, assume we have a core specification that contains the following JCR:

```
#ruleset-id com.example.core
core { core-item1, core-item2 }
more { core-item3, core-item4 }
...
```

And a subsequently defined extension with the following JCR:

```
#ruleset-id com.example.extension
#import com.example.core as core
extension @{augments core.core core.more} ext-item1
```

The resultant core specification is treated as:

```
#ruleset-id com.example.core
core { core-item1, core-item2, __alias1.extension }
more { core-item3, core-item4, __alias1.extension }
...
```

where '__alias1' is conceptually an automatically created alias that aliases 'com.example.extension'.

Because multiple @{augments} annotations may specify the same target-rule-name, there can be no control over the order the augmentations are given in the target rule. Hence the specified target-rule-names are only allowed to correspond to (unordered) objects, unordered arrays, and value choices.

If the non-nested rules in the target rule are all combined using the choice combiner, then the augmenting rule is also combined using the choice combiner. If the non-nested rules in the target rule are all combined using the sequence combiner, then the augmenting rule is also combined using the sequence combiner. If the non-nested rules in the target rule use a combination of the choice combiner and sequence combiner, then the existing rules within the target group are logically nested within a group and the augmenting rule is combined using the sequence combiner. For example, a target rule initially containing the following definition:

```
core { core-item1, core-item2 | core-item3 }
```

would be treated as follows after being augmented:

```
core { (core-item1, core-item2 | core-item3), __alias1.extension }
```

If it is desired to add more than one rule to a target rule then the augmenting rule can specify a group, for example:

```
extension @{augments core.core core.more} (ext-item1, ext-item2)
```

Appendix D. JCR Implementations

The following implementations, [ARIN_JCR_VALIDATOR] and [CODALOGIC_JCR_VALIDATOR] have influenced the development of this document.

Authors' Addresses

Andrew Lee Newton
American Registry for Internet Numbers
3635 Concorde Parkway
Chantilly, VA 20151
US

Email: andy@arin.net
URI: http://www.arin.net

Pete Cordell
Codalogic
PO Box 30
Ipswich IP5 2WY
UK

Email: pete.cordell@codalogic.com
URI: http://www.codalogic.com