Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 1, 2018

A. Newton
ARIN
P. Cordell
Codalogic

September 28, 2017

A Language for Rules Describing JSON Content
draft-newton-json-content-rules-09

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

Copyright Notice

Table of Contents

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 }
```

Figure 1

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 }
```

Figure 2

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.. }
```

Figure 3

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
}
```

Figure 4

The following JCR describes objects like it.

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

Figure 5

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..
```

Figure 6

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
```

Figure 7

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

All figures used in this specification are available here [1].

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. Rulesets may also contain comments, blank lines, and directives that apply to the processing of a ruleset.

Rules are composed of two parts, an optional rule name and a rule specification. A rule specification can be either a type specification or a member specification. A member specification consists of a member name specification and a type specification.

A type specification is used to specify constraints on a superset of a JSON value (e.g. number / string / object / array etc.). In addition to defining primitive types (such as string or integer), array types, and object types, type specifications may define the JCR specific concept of group types.

Type specifications corresponding to arrays, objects and groups may be composed of other rule specifications.

A member specification is used to specify constraints on a JSON member (i.e. members of a JSON object).

Rules with rule name assignments may be referenced in place of type specifications and member specifications.

Rules may be defined across line boundaries and there is no line continuation syntax.

Any rule consisting only of a type specification is considered a root rule. Unless otherwise specified, all the root rules of a ruleset are evaluated against a JSON instance or document.

Putting it all together, Figure 9 describes the JSON in Figure 8.

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 8

Rules describing Figure 8

```
; the root of the JSON instance is an object
; this root rule describes that object
{

  ; the object specification contains
  ; one member specification
  "Image" : {

    ; $width and $height are defined below
    $width,
    $height,

    ; "Title" member specification
    "Title" :string,

    ; "Thumbnail" member specification, which
    ; defines an object
    "Thumbnail":  {

      ; $width and $height are re-used again
      $width, $height,

      "Url" :uri
    },

    ; "IDs" member that is an array of
    ; one ore more integers
    "IDs" : [ integer * ]

  }
}

; The definitions of the rules $width and $height
$width  = "Width" : 0..1280
$height = "Height" : 0..1024
```

Figure 9

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 the same as comments in ABNF [RFC4234]. They start with a semi-colon (';') and continue to the end of the line.

4. Rules

Rules have two main components, an optional rule name assignment and a type or member specification.

Type specifications define arrays, objects, etc... of JSON and may reference other rules using rule names. Most type specifications can be defined with repetitions for specifying the frequency of the type being defined. In addition to the type specifications describing JSON types, there is an additional group specification for grouping types.

Member specifications define members of JSON objects, and are composed of a member name specification and either a type specification or a rule name referencing a type specification.

Rules may also contain annotations which may affect the evaluation of all or part of a rule. Rules without a rule name assignment are considered root rules, though rules with a rule name assignment can be considered a root rule with the appropriate annotation.

Type specifications, depending on their type, can contain zero or more other specifications or rule names. For example, an object specification might contain multiple member specifications or rule names that resolve to member specifications or a mixture of member specifications and rule names. For the purposes of this document, specifications and rule names composing other specifications are called subordinate components.

4.1. Rule Names and Assignments

Rule names are signified with the dollar character ('$'), which is not part of the rule name itself. Rule names have two components, an optional ruleset identifier alias and a local rule name.

Local 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 ('_'). Local rule names are case sensitive, and must be unique within a ruleset (that is, no two rule name assignments may use the same local rule name).

Ruleset identifier aliases enable referencing rules from another ruleset. They are not allowed in rule name assignments, and only found in rule names referencing other rules. Ruleset identifiers must start with an alphabetic character and contain no whitespace. Ruleset identifiers are case sensitive. Simple use cases of JCR will most likely not use ruleset identifiers.

In Figure 10 below, "http://ietf.org/rfcYYYY.JCR" and "http://ietf.org/rfcXXXX.JCR" are ruleset identifiers and "rfcXXXX" is a ruleset identifier alias.

```
# ruleset-id http://ietf.org/rfcYYYY.JCR
# import http://ietf.org/rfcXXXX.JCR as rfcXXXX
$my_encodings  = ( "mythic" | "magic" )
$all_encodings = ( $rfcXXXX.encodings | $my_encodings )
```

Figure 10

There are two forms of rule name assignments: assignments of primitive types and assignments of all other types. Rule name assignments to primitive type specifications separate the rule name from the type specification with the character sequence ’=:’, whereas rule name assignments for all other type specifications only require the separation using the ’=’ character.

```
;rule name assignments for primitive types
$foo         =: "foo"
$some_string =: string

;rule name assignments for arrays
$bar = [ integer, integer, integer ]

;rule name assignement for objects
$bob = { "bar" : $bar, "foo" : $foo }
```

Figure 11

This is the one little "gotcha" in JCR. This syntax is necessary so that JCR parsers may readily distinguish between rule name assignments involving string and regular expressions primitive types and member names of member specifications.

4.2. Annotations

Annotations may appear before a rule name assignment, before a type or member specification, or before a rule name contained within a type specification. In each place, there may be zero or more annotations. Each annotation begins with the character sequence "@{" and ends with "}". The following is an example of a type specification with the not annotation (explained in Section 4.14):

```
@{not} [ "fruits", "vegatables" ]
```

Figure 12

This specification defines the annotations "root", "not", and "unordered", but other annotations may be defined for other purposes.

4.3. Starting Points and Root Rules

Evaluation of a JSON instance or document against a ruleset begins with the evaluation of a root rule or set of root rules. If no root rule (or rules) is specified locally at runtime, the set of root rules specified in the ruleset are evaluated. The order of evaluation is undefined.

The set of root rules specified in a ruleset is composed of all rules without a rule name assignment and all rules annotated with the "@{root}" annotation.

The "@{root}" annotation may either appear before a rule name assignment or before a type definition. It is ignored if present before referenced rule name inside of a type specification.

4.4. Type Specifications

The syntax of each type of type specifications varies depending on the type:

```
      ; primitive types can be string
      ; or number literals
      ; or number ranges
      "foo"
      2
      1..10

      ; primitive types can also be more generalized types
      string
      integer

      ; primitive type rules may be named
      $my_int =: 12

      ; member specifications consist of a member name
      ; followed by a colon and then followed by another
      ; type specification or a rule name
      ; (example shown with a rule name assignment)
      $mem1 = "bar" : "baz"
      $mem2 = "fizz" : $my_int

      ; member names may either be quoted strings
      ; or regular expressions
      ; (example shown with a rule name assignment)
      $mem3 = /^dev[0-9]$/ : 0..4096

      ; object specifications start and end with "curly braces"
      ; object specifications contain zero
      ; or more member specifications
      ; or rule names which reference a member specification
      { $mem1, "foo" : "fuzz", "fizz" : $my_int }

      ; array specifications start and end with square brackets
      ; array specifications contain zero
      ; or more non-member type specifications
      [ 1, 2, 3, $my_int ]

      ; finally, group specifications start and end with parenthesis
      ; groups contain other type specifications
      ( [ integer, integer], $rule1 )
      $rule1 = [ string, string ]
```

Figure 13

4.5. Primitive Specifications

Primitive type specifications define content for JSON numbers, booleans, strings, and null.

4.5.1. Numbers, Booleans and Null

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

```
true
false
boolean
null
```

Figure 14

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

```
integer
float
double
```

Figure 15

The keyword ’float’ represents a single precision IEEE-754 floating point number represented in decimal. The keyword ’double’ represents a double precision IEEE-754 floating point number represented in decimal format.

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:

```
  n
  n..m
   ..m
  n..
n.f
n.f..m.f
   ..m.f
n.f..
```

Figure 16

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
```

Figure 17

Integers may also be specified as ranges using bit lengths preceded by the 'int' or 'uint' words (i.e. 'int8', 'uint16'). The 'int' prefix specifies the integer as being signed whereas the 'uint' prefix specifies the integer as being unsigned.

```
; 0..255
uint8

; -32768..32767
int16

; 0..65535
uint16

; -9223372036854775808..9223372036854775807
int64

; 0..18446744073709551615
uint64
```

Figure 18

4.5.2. Strings

JCR provides a large number of data types to define the contents of JSON strings. Generically, a string may be specified using the word 'string'. String literals may be specified using a double quote character followed by the literal content followed by another double quote. And regular expressions may be specified by enclosing a regular expression within the forward slash ('/') character.

```
                          ; any string
                          string

                          ; a string literal
                          "she sells sea shells"

                          ; a regular expression
                          /^she sells .*/
```

Figure 19

Regular expressions are not implicitly anchored and therefore must be explicitly anchored if necessary.

A string can be specified as a URI [RFC3986] using the word ’uri’, but also may be more narrowly scoped to a URI of a specific scheme. Specific URI schemes are specified with the word ’uri’ followed by two period characters (’..’) followed by the URI scheme.

```
                     ; any URI
                     uri

                     ;a URI narrowed for an HTTPS uri
                     uri..https
```

Figure 20

IP addresses may be specified with either the word ’ipv4’ for IPv4 addresses [RFC1166] or the word ’ipv6’ for IPv6 addresses [RFC5952]. Fully qualified A-label and U-label domain names may be specified with the words ’fqdn’ and ’idn’.

Dates and time can be specified as formats found in RFC 3339 [RFC3339]. The word ’date’ corresponds to the full-date ABNF rule, the word ’time’ corresponds to the full-time ABNF rule, and the word ’datetime’ corresponds to the ’date-time’ ABNF rule.

Email addresses formatted according to RFC 5322 [RFC5322] may be specified using the ’email’ word, and E.123 phone numbers may be specified using the word ’phone’.

```
;IP addresses
ipv4
ipv6
ipaddr

;domain names
fqdn
idn

; RFC 3339 full-date
date
; RFC 3339 full-time
time
; RFC 3339 date-time
datetime

; RFC 5322 email address
email

; phone number
phone
```

Figure 21

Binary data can be specified in string form using the encodings specified in RFC 4648 [RFC4648]. The word ’hex’ corresponds to base16, while ’base32’, ’base32hex’, ’base64’, and ’base64url’ correspond with their RFC 4648 counterparts accordingly.

```
; RFC 4648 base16
hex

; RFC 4648 base32
base32

; RFC 4648 base32hex
base32hex

; RFC 4648 base64
base64

; RFC 4648 base64url
base64url
```

Figure 22

4.6. Any Type

It is possible to specify that a value can be of any type allowable by JSON using the word ’any’. The ’any’ type specifies any primitive type, array, or object.

4.7. Member Specifications

Member specifications define members of JSON objects. Unlike other type specifications, member specifications cannot be root rules and must be part of an object specification or preceded by a rule name assignment.

Member specifications consist of a member name specification followed by a colon character (’:’) followed by either a subordinate component, which is either a rule name or a primitive, object, array, or group specification. Member name specifications can be given either as a quoted string using double quotes or as a regular expression using forward slash (’/’) characters. Regular expressions are not implicitly anchored and therefore must have explicit anchors if needed.

```
;member name will exactly match "locationURI"
$location_uri = "locationURI" : uri

;member name will match "eth0", "eth1", ... "eth9"
$iface_mappings = /^eth[0-9]$/  : ipv4
```

Figure 23

4.8. Object Specifications

Object specifications define JSON objects and are composed of zero or more subordinate components, each of which can be either a rule name, member specification, or group specification. The subordinate components are enclosed at the start with a left curly brace character (’{’) and at the end with a right curly brace character (’}’).

Evaluation of the subordinate components of object specifications is as follows:

- o No order is implied for the members of the object being evaluated.
- o Subordinate components of the object specification are evaluated in the order they appear.

- Each member of the object being evaluated can only match one subordinate component.

- Any members not matched against a subordinate component are ignored.

The following examples illustrate matching of JSON objects to JCR object specifications.

As order is not implied for the members of objects under evaluation, the following rule will match the JSON in Figure 25 and Figure 26.

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

Figure 24

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

Figure 25

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

Figure 26

Because subordinate components of an object specification are evaluated in the order in which they are specified (i.e. left to right, top to bottom) and object members can only match one subordinate component of an object specification, the rule o1 below will not match against the JSON in Figure 28 but the rule o2 below will match it.

```
          ; zero or more members that match "p0", "p1", etc
          ; and a member that matches "p1"
          $o1 = { /^p\d+$/ : integer *, "p1" : integer }

          ; a member that matches "p1" and
          ; zero or more members that match "p0", "p1", etc
          $o2 = { "p1" : integer, /^p\d+$/ : integer * }
```

The first subordinate of rule o1 specifies that an object can have zero or more members (that is the meaning of "*", see Section 4.13) where the member name is the letter 'p' followed by a number (e.g. "p0", "p1", "p2"), and the second rule specifies a member with the exact member name of "p1". Rule o2 has the exact same member specifications but in the opposite order. Figure 28 does not match rule o1 because all of the members match the first subordinate rule leaving none to match the second subordinate rule. However, rule o2 does match because the first subordinate rule matches only one member of the JSON object allowing the second subordinate rule to match the other member of the JSON object.

Figure 27

```
{ "p0" : 1, "p1" : 2 }
```

Figure 28

As stated above, 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 a rule specifying that no extra members are expected, the @{not} annotation (see Section 4.14) may be used with a "match-all" regular expression as the last subordinate component of the object specification.

The following rule will match the JSON object in Figure 30 but will not match the JSON object in Figure 31.

```
{ "foo" : 1, "bar" : 2, @{not} // : any + }
```

Figure 29

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

Figure 30

```
{ "foo" : 1, "bar" : 2, "baz" : 3 }
```

Figure 31

This works because subordinate components are evaluated in the order they appear in the object rule, and the last component accepts any member with any type but fails to validate if one or more of those components are found due to the @{not} annotation.

4.9. Array Specifications

Array specifications define JSON arrays and are composed of zero or more subordinate components, each of which can either be a rule name or a primitive, array, object or group specification. The subordinate components are enclosed at the start with a left square brace character (’[’) and at the end with a right square brace character (’]’).

Evaluation of the subordinate components of array specifications is as follows:

- The order of array items is implied unless the @{unordered} annotation is present.
- Subordinate components of the array specification are evaluated in the order they appear.
- Each item of the array being evaluated can only match one subordinate component of the array specification.
- If any items of the array are not matched, then the array does not match the array specification.

These rules are further explained in the examples below.

```
[ 0..1024, 0..980 ]
```

Figure 32

Unlike object specifications, order is implied in array specifications by default. That is, the first subordinate component will match the first element of the array, the second subordinate component will match the second element of the array, and so on.

Take for example the following ruleset:

```
; the first element of the array is to be a string
; the second element of the array is to be an integer
$a1 = [ string, integer ]

; the first element of the array is to be an integer
; the second element of the array is to be a string
$a2 = [ integer, string ]
```

Figure 33

It defines two rules, a1 and a2. The array in the following JSON will not match a1, but will match a2.

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

Figure 34

If an array has more elements than can be matched from the array specification, the array does not match the array specification. Or stated differently, an array with unmatched elements does not validate. Using the example array rule a2 from above, the following array does not match because the last element of the array does not match any subordinate component:

```
[ 24, "Bob Smurd", "http://example.com/bob_smurd" ]
```

Figure 35

To allow an array to contain any value after guaranteeing that it contains the necessary items, the last subordinate component of the array specification should accept any item:

```
; the first element of the array is to be an integer
; the second element of the array is to be a string
; anything else can follow
$a3 = [ integer, string, any * ]
```

The JSON array in Figure 35 will validate against the a3 rule in this example.

Figure 36

4.9.1. Unordered Array Specifications

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

In the ruleset below, a1 and a2 have the same subordinate components given in the same order. a2 is annotated with the @{unordered} annotation.

```
$a1 =              [ string, integer ]
$a2 = @{unordered} [ string, integer ]
```

Figure 37

The JSON array below does not match a1 but does match a2.

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

Figure 38

Like ordered array specifications, the subordinate components in an unordered array specification 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 specification.

Finally, like ordered array specifications, unordered array specifications also require that all elements of the array be matched by a subordinate component. If the array has more elements than can be matched, the array does not match the array specification.

4.10. Group Specifications

Unlike the other type specifications, group specifications have no direct tie with JSON syntax. Group specifications simply group together their subordinate components. Group specifications enclose one or more subordinate components with the parenthesis characters.

Group specifications and any nesting of group specifications, must conform to the allowable set of type specifications of the type specifications in which they are contained. For example, a group specification inside of an array specification may not contain a member specification since member specifications are not allowed as direct subordinates of array specifications (arrays contain values, not object members in JSON). Likewise, a group specification referenced inside an object specification must only contain member specifications (JSON objects may only contain object members).

The following is an example of a group specification:

```
          $the_bradys = [ $parents, $children ]

          $children = ( "Greg", "Marsha", "Bobby", "Jan" )

          $parents = ( "Mike", "Carol" )
```

Figure 39

Like the subordinate components of array and object specifications, the subordinate components of a group specification are evaluated in the order they appear.

4.11. Ordered and Unordered Groups in Arrays

Section 4.9.1 specifies that arrays can be evaluated by the order of the items in the array or can be evaluated without order. Section 4.10 specifies that arrays may have group rules as subordinate components.

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

4.12. Sequence and Choice Combinations in Array, Object, and Group Specifications

Combinations of subordinate components in array, object, and group specifications can be specified as either a sequence ("and") or a choice ("or"). A sequence is a subordinate component followed by the comma character (’,’) followed by another subordinate component. A choice is a subordinate component followed by a pipe character (’|’) followed by another subordinate component.

```
                    ; sequence ("and")
                    [ "this" , "that" ]

                    ; choice ("or")
                    [ "this" | "that" ]
```

Figure 40

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

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

Figure 41

The example above should be expressed as:

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

Figure 42

> NOTE: A future specification will clarify the choice ('|') operation as inclusive or, exclusive or ("xor") or otherwise. At present readers should assume the choice ('|') operator is an inclusive or. However, for objects and unordered arrays that is not ideal, nor is xor. We are in the process of defining an algorithm to "rewrite" choices of rules for use with inclusive or which is more suitable for the data model of JSON.

4.13. Repetition in Array, Object, and Group Specifications

Evaluation of subordinate components in array, object, and group specifications may be succeeded by a repetition expression denoting how many times the subordinate component should be evaluated. Repetition expressions are specified using a Kleene symbol ('?', '+', or '*') or with the '*' symbol succeeded by specific minimum and/or maximum values, each being non-negative integers. Repetition expressions may also be appended with a step expression, which is the '%' symbol followed by a positive integer.

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 two period characters ('..') followed by the maximum, with either the minimum or maximum being optional. When the minimum is not explicitly specified, it is assumed to be zero. When the maximum is not explicitly specified, it is assumed to be positive infinity.

```
; exactly 2 octets
$word = [ $octet *2 ]
$octet =: int8

; 1 to 13 name servers
[ $name_servers *1..13 ]
$name_servers =: fqdn

; 0 to 99 ethernet addresses
{ /^eth.*/ : $mac_addr *..99 }
$mac_addr =: hex

; four or more bytes
[ $octet *4.. ]
```

Figure 43

The allowable Kleene operators are the question mark character ('?') which specifies zero or one (i.e. optional), the plus character ('+') which specifies one or more, and the asterisk character ('*') which specifies zero or more.

```
; age is optional
{ "name" : string, "age" : integer ? }

; zero or more errors
$error_set = ( string * )

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

Figure 44

A repetition step expression may follow a minimum to maximum expression or the zero or more Kleene operator or the one or more Kleene operator.

- o When the repetition step follows a minimum to maximum expression or the zero or more Kleene operator ('*'), it specifies that the total number of repetitions present in the JSON instance being validated minus the minimum repetition value must be a multiple of the repetition step (e.g. the total repetitions minus the minimum repetition value must be divisible by the step value with a remainder of zero).

- When the repetition step follows a one or more Kleene operator ('+'), the minimum repetition value is set equal to the repetition step value and the total number of repetitions minus the step value must be a multiple of the repetition step value.

The following is an example for repetition steps in repetition expressions.

```
; there must be at least 2 name servers
; there may be no more than 12 name servers
; there must be an even number of name servers
; e.g. 2,4,6,8,10,12
[ $name_servers *2..12%2 ]
$name_servers =: fqdn

; minimum is zero
; maximum is 100
; must be an even number
{ /^eth.*/ : $mac_addr *..100%2 }
$mac_addr =: hex

; at least 32 octets
; must be be in groups of 16
; e.g. 32, 48, 64 etc
[ $octet *32..%16 ]
$octet =: int8

; if there are to be error sets,
; their number must be divisible by 4
; e.g. 0, 4, 8, 12 etc
$error_set = ( string *%4 )

; Throws of a pair of dice must be divisible by 2
; e.g. 2, 4, 6 etc
$dice_throws = ( 1..6 +%2 )
```

Figure 45

4.14. Negating Evaluation

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

```
; match anything that isn't the integer 2
$not_two = [ @{not} 2 ]

; error if one of the status values is "fail"
$status = @{not} @{unordered} [ "fail", string * ]
```

Figure 46

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 parameter_1 parameter_2 ...
```

Figure 47

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

```
#{ directive_name
    parameter_1 paramter_2
    parameter_3
    ...
}
```

Figure 48

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 0.7
```

Figure 49

The major.minor number signifying compliance with this document is "0.7". Upon publication of this specification as an IETF proposed standard, it will be "1.0".

```
# jcr-version 1.0
```

Figure 50

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

This directive may have optional extension identifiers following the version number. Each extension identifiers is preceded by the plus ('+') character and separated by white space. The format of extension identifiers is specific to the extension, but it is recommended that they are terminated by a version number.

```
# jcr-version 1.0 +co-constraints-1.2 +jcr-doc-1.0
```

Figure 51

5.2. ruleset-id

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

```
# ruleset-id identifier
```

Figure 52

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 opaque, 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.

The following is an example:

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

Figure 53

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

```
{ // : string }
```

Figure 54

The JSON below matches the above rule.

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

Figure 55

Likewise, the JSON below also matches the same rule.

```
{ "fuzz" : "bazz" }
```

Figure 56

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

```
{ // : any }
```

Figure 57

The above rule matches not only the two JSON objects above, but the JSON object below.

```
{ "fuzz" : 1234 }
```

Figure 58

6.2. Lists of Values

Group specifications may be used to create enumerated lists of primitive data types, because primitive specifications may contain a group specification, which may have multiple primitive specifications. Because a primitive specification must resolve to a single data type, the group specification must only contain choice combinations.

Consider the following examples:

```
; either an IPv4 or IPv6 adress
$address =: ( ipv4 | ipv6 )

; allowable fruits
$fruits =: ( "apple" | "banana" | "pear" )
```

Figure 59

6.3. Groups in Arrays

Groups may be a subordinate component of array specifications:

```
[ ( ipv4 | ipv6 ), integer ]
```

Figure 60

Unlike primitive specifications, subordinate group specifications in array specifications may have sequence combinations and contain any type specification.

```
; a group in an array
[ ( $first_name, $middle_name ?, $last_name ), $age ]

; a group referenced from an array
[ $name, $age ]
$name = ( $first_name, $middle_name ?, $last_name )

$first_name =: string
$middle_name =: string
$last_name =: string
$age =: 0..
```

Figure 61

6.4. Groups in Objects

Groups may be a subordinate component of object specifications: Subordinate group specifications in object specifications may have sequence combinations but must only contain member specifications.

```
; a group in an object
{ ( $title, $date, $author ), $paragraph + }

; a group referenced from an object
{ $front_matter, $paragraph + }
$front_matter = ( $title, $date, $author )

$title = "title" : string
$date = "date" : date
$author = "author" : [ string * ]
$paragraph = /p[0-9]*/ : string
```

Figure 62

NOTE: A future specification will clarify the choice ('|') operation as inclusive or, exclusive or ("xor") or otherwise. At present readers should assume the choice ('|') operator is an inclusive or. We are in the process of defining an algorithm to "rewrite" choices of rules for use with inclusive or which is more suitable for the data model of JSON. Such a change will impact the guidance given below.

When using groups to use both sequences and choices of member specifications, consideration must be given to the processing of object specifications where by unmatched member specifications are ignored (see Figure 23).

A casual reading of this rule might lead a reader to believe that the JSON object in Figure 64 would not match, however it does because the extra member (either "foo" or "baz") is not matched but is ignored.

```
{ "bar":string, ( "foo":integer | "baz":string ) }
```

Figure 63

```
{ "bar":"thing", "foo":2, "baz": "thingy" }
```

Figure 64

The rule in Figure 63 must be modified to either match all extra rules, as in Figure 65, or the logic of the rules must be rewritten to explicitly negate the presence of the unwanted members, as in Figure 66.

```
 { "bar":string, ( "foo":integer | "baz":string ), @{not} //:any + }
```

Figure 65

```
{ "bar":string,
  ( ( "foo":integer , @{not} "baz":string ) |
    ( "baz":string , @{not} "foo":integer )
) }
```

Figure 66

6.5. Group Rules as Macros

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

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

Figure 67

6.6. 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 below, 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 68

6.7. Subordinate Dependencies

In object and array specifications, there may be situations in which it is necessary to condition the existence of a subordinate component on the existence of a sibling subordinate component. In other words, example_two should only be evaluated if example_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 or item is present.

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

```
; $referrer_uri can only be present if
; $location_uri is present
{ ( $location_uri, $referrer_uri? )? }

$location_uri = "locationURI" : uri
$referrer_uri = "referrerURI" : uri
```

Figure 69

7. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7492] . The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7492] , "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

7.1. JCR Validator

The JCR Validator, written in Ruby, currently implements all portions of this specification, and has been used extensively to prototype various aspects of JCR under consideration. It’s development has gone hand-in-hand with this specification.

This software is primarily produced by the American Registry for Internet Numbers (ARIN) and freely distributable under the ISC license.

Source code for this software is available on GitHub at <https://github.com/arineng/jcrvalidator>. This software is also easily obtained as a Ruby Gem through the Ruby Gem system.

7.2. Codalogic JCR Parser

The Codalogic JCR Parser is a C++ implementation of a JCR parsing engine, and is a work in progress. It is targeted for the Windows platform.

This software is produced by Codalogic Ltd and freely distributable under the Gnu LGPL v3 license.

Source code is availabe on GitHub at <https://github.com/codalogic/cl-jcr-parser>.

7.3. JCR Java

JCR Java is a work in progress and currently only implements the parsing of JCR rulesets according to the ABNF using a custom parsing framework.

This software is produced by the American Registry for Internet Numbers (ARIN) and freely distributable under the MIT license.

Source code is available on BitBucket at <https://bitbucket.org/anewton_1998/jcr_java>.

8. ABNF Syntax

The following ABNF describes the syntax for JSON Content Rules. A text file containing these ABNF rules can be downloaded from [JCR_ABNF].

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

sp-cmt           = spaces / comment
spaces           = 1*( WSP / CR / LF )
DSPs             = ; Directive spaces
                   1*WSP /     ; When in one-line directive
                   1*sp-cmt   ; When in muti-line directive
comment          = ";" *comment-char comment-end-char
comment-char     = HTAB / %x20-10FFFF
                   ; Any char other than CR / LF
comment-end-char = CR / LF

directive        = "#" (one-line-directive / multi-line-directive)
one-line-directive = [ DSPs ]
```

```
                   (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 DSPs major-version
                   "." minor-version
                   *( DSPs "+" [ DSPs ] extension-id )
major-version    = non-neg-integer
minor-version    = non-neg-integer
extension-id     = ALPHA *not-space
ruleset-id-d     = ruleset-id-kw DSPs ruleset-id
import-d         = import-kw DSPs ruleset-id
                   [ DSPs as-kw DSPs 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
                   [ 1*sp-cmt 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             = annotations "$" rule-name *sp-cmt
                   "=" *sp-cmt rule-def

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

rule-def         = member-rule / type-designator rule-def-type-rule /
                   array-rule / object-rule / group-rule /
                   target-rule-name
type-designator  = type-kw 1*sp-cmt / ":" *sp-cmt
```

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

annotations      = *( "@{" *sp-cmt annotation-set *sp-cmt "}"
                   *sp-cmt )
annotation-set   = not-annotation / unordered-annotation /
                   root-annotation / tbd-annotation
not-annotation   = not-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 primitive-def
primitive-def    = string-type / string-range / string-value /
                   null-type / boolean-type / true-value /
                   false-value / double-type / float-type /
                   float-range / float-value /
                   integer-type / integer-range / integer-value /
                   sized-int-type / sized-uint-type / ipv4-type /
                   ipv6-type / ipaddr-type / fqdn-type / idn-type /
                   uri-type / phone-type / email-type /
                   datetime-type / date-type / time-type /
                   hex-type / base32hex-type / base32-type /
                   base64url-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
double-type      = double-kw
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
 sized-int-type   = int-kw pos-integer
 sized-uint-type  = uint-kw pos-integer
 ipv4-type        = ipv4-kw
 ipv6-type        = ipv6-kw
 ipaddr-type      = ipaddr-kw
 fqdn-type        = fqdn-kw
 idn-type         = idn-kw
 uri-type         = uri-kw [ ".." uri-scheme ]
 phone-type       = phone-kw
 email-type       = email-kw
 datetime-type    = datetime-kw
 date-type        = date-kw
 time-type        = time-kw
 hex-type         = hex-kw
 base32hex-type   = base32hex-kw
 base32-type      = base32-kw
 base64url-type   = base64url-kw
 base64-type      = base64-kw
 any              = any-kw

 object-rule      = annotations "{" *sp-cmt
                    [ object-items *sp-cmt ] "}"
 object-items     = object-item [ 1*( sequence-combiner object-item ) /
                    1*( choice-combiner object-item ) ]
 object-item      = object-item-types *sp-cmt [ repetition *sp-cmt ]
 object-item-types = object-group / member-rule / target-rule-name
 object-group     = annotations "(" *sp-cmt [ object-items *sp-cmt ] ")"

 array-rule       = annotations "[" *sp-cmt [ array-items *sp-cmt ] "]"
 array-items      = array-item [ 1*( sequence-combiner array-item ) /
                    1*( choice-combiner array-item ) ]
 array-item       = array-item-types *sp-cmt [ repetition *sp-cmt ]
 array-item-types = array-group / type-rule / explicit-type-choice
 array-group      = annotations "(" *sp-cmt [ array-items *sp-cmt ] ")"

 group-rule       = annotations "(" *sp-cmt [ group-items *sp-cmt ] ")"
 group-items      = group-item [ 1*( sequence-combiner group-item ) /
                    1*( choice-combiner group-item ) ]
 group-item       = group-item-types *sp-cmt [ repetition *sp-cmt ]
 group-item-types = group-group / member-rule /
                    type-rule / explicit-type-choice
 group-group      = group-rule
```

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

repetition        = optional / one-or-more /
                    repetition-range / zero-or-more
optional          = "?"
one-or-more       = "+" [ repetition-step ]
zero-or-more      = "*" [ repetition-step ]
repetition-range = "*" *sp-cmt (
                    min-max-repetition / min-repetition /
                    max-repetition / specific-repetition )
min-max-repetition = min-repeat ".." max-repeat
                    [ repetition-step ]
min-repetition    = min-repeat ".." [ repetition-step ]
max-repetition    = ".."  max-repeat [ repetition-step ]
min-repeat        = non-neg-integer
max-repeat        = non-neg-integer
specific-repetition = non-neg-integer
repetition-step   = "%" step-size
step-size         = non-neg-integer

integer           = "0" / ["-"] pos-integer
non-neg-integer   = "0" / pos-integer
pos-integer       = digit1-9 *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-scheme       = 1*ALPHA

;; Keywords
any-kw           = %x61.6E.79                         ; "any"
as-kw            = %x61.73                            ; "as"
base32-kw        = %x62.61.73.65.33.32                ; "base32"
base32hex-kw     = %x62.61.73.65.33.32.68.65.78       ; "base32hex"
base64-kw        = %x62.61.73.65.36.34                ; "base64"
base64url-kw     = %x62.61.73.65.36.34.75.72.6C       ; "base64url"
boolean-kw       = %x62.6F.6F.6C.65.61.6E             ; "boolean"
date-kw          = %x64.61.74.65                      ; "date"
datetime-kw      = %x64.61.74.65.74.69.6D.65          ; "datetime"
double-kw        = %x64.6F.75.62.6C.65                ; "double"
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"
hex-kw           = %x68.65.78                         ; "hex"
idn-kw           = %x69.64.6E                         ; "idn"
import-kw        = %x69.6D.70.6F.72.74                ; "import"
int-kw           = %x69.6E.74                         ; "int"
integer-kw       = %x69.6E.74.65.67.65.72             ; "integer"
ipaddr-kw        = %x69.70.61.64.64.72                ; "ipaddr"
ipv4-kw          = %x69.70.76.34                      ; "ipv4"
ipv6-kw          = %x69.70.76.36                      ; "ipv6"
jcr-version-kw   = %x6A.63.72.2D.76.65.72.73.69.6F.6E ; "jcr-version"
not-kw           = %x6E.6F.74                         ; "not"
null-kw          = %x6E.75.6C.6C                      ; "null"
phone-kw         = %x70.68.6F.6E.65                   ; "phone"
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"
time-kw          = %x74.69.6D.65                      ; "time"
true-kw          = %x74.72.75.65                      ; "true"
type-kw          = %x74.79.70.65                      ; "type"
uint-kw          = %x75.69.6E.74                      ; "uint"
unordered-kw     = %x75.6E.6F.72.64.65.72.65.64    ; "unordered"
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
```

Figure 70: ABNF for JSON Content Rules

9. Acknowledgements

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

10. References

10.1. Normative References

[JCR_ABNF]
Newton, A. and P. Cordell, "ABNF for JSON Content Rules", <https://raw.githubusercontent.com/arineng/jcr/master/jcr-abnf.txt>.

[RFC1166] Kirkpatrick, S., Stahl, M., and M. Recker, "Internet numbers", RFC 1166, DOI 10.17487/RFC1166, July 1990, <https://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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc4234>.

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

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://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, <https://www.rfc-editor.org/info/rfc5952>.

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

10.2. 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.

[RFC7492] Bhatia, M., Zhang, D., and M. Jethanandani, "Analysis of Bidirectional Forwarding Detection (BFD) Security According to the Keying and Authentication for Routing Protocols (KARP) Design Guidelines", RFC 7492, DOI 10.17487/RFC7492, March 2015, <https://www.rfc-editor.org/info/rfc7492>.

10.3. URIs

[1] https://github.com/arineng/jcr/tree/master/figs

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 * ]
```

Figure 71

Consider the specification where this rule is found does not define the values but references an extensible list of possible values updated independently of the specification, such as in an IANA registry.

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:

This rule will evaluate positively with the JSON in Figure 73

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

Figure 72

```
[ "submitted", "validated", "accepted" ]
```

Figure 73

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

This rule will fail to evaluate the JSON in Figure 75 thus signaling a problem.

```
$statuses = @{unordered} @{not} [ "denied" + , string * ]
```

Figure 74

```
[ "submitted", "validated", "denied" ]
```

Figure 75

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. Changes from -07 and -08

This revision of the document makes no substantive changes to any parts of the specification. Some of the ABNF has been updated to more correctly allow group rules, and other small change have been made to the ABNF to make it simpler.

Authors' Addresses

Andrew Lee Newton
American Registry for Internet Numbers
PO Box 232290
Centreville, VA 20120
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