Workgroup: Network Working Group
Internet-Draft:
draft-bormann-cbor-cddl-control-02
Published: 2 September 2020
Intended Status: Informational
Expires: 6 March 2021
Authors: C. Bormann
Universität Bremen TZI

Additional Control Operators for CDDL

Abstract

The Concise Data Definition Language (CDDL), standardized in RFC 8610, provides "control operators" as its main language extension point.

The present document defines a number of control operators that did not make it into RFC 8610: .cat/.plus for the construction of constants, .abnf/.abnfb for including ABNF (RFC 5234/RFC 7405) in CDDL specifications, and .feature for indicating the use of a non-basic feature in an instance.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The Concise Data Definition Language (CDDL), standardized in RFC 8610, provides "control operators" as its main language extension point.

The present document defines a number of control operators that did not make it into RFC 8610:

Name	Purpose
.cat	String Concatenation
.plus	Numeric addition
.abnf	ABNF in CDDL (text strings)
.abnfb	ABNF in CDDL (byte strings)
.feature	Detecting feature use in extension points

Table 1: New control operators in this document

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses terminology from [RFC8610]. In particular, with respect to control operators, "target" refers to the left hand side operand, and "controller" to the right hand side operand.

2. Computed Literals

CDDL as defined in [RFC8610] does not have any mechanisms to compute literals. As an 80 % solution, this specification adds two control operators: .cat for string concatenation, and .plus for numeric addition.

2.1. String Concatenation

It is often useful to be able to compose string literals out of component literals defined in different places in the specification.

The .cat control identifies a string that is built from a concatenation of the target and the controller. As targets and controllers are types, the resulting type is formally the cross-product of the two types, although not all tools may be able to work with non-unique targets or controllers.

Target and controller MUST be strings. The result of the operation has the type of the target. The concatenation is performed on the bytes in both strings. If the target is a text string, the result of that concatenation MUST be valid UTF-8.

```
a = "foo" .cat '
  bar
  baz
'
; on a system where the newline is \n, is the same string as:
b = "foo\n  bar\n  baz\n"
```

Figure 1: Example: concatenation of text and byte string

The example in Figure 1 builds a text string named a out of concatenating the target text string "foo" and the controller byte string entered in a text form byte string literal. (This particular idiom is useful when the text string contains newlines, which, as shown in the example for b, may be harder to read when entered in the format that the pure CDDL text string notation inherits from JSON.)

2.2. Numeric Addition

In many cases in a specification, numbers are needed relative to a base number. The .plus control identifies a number that is constructed by adding the numeric values of the target and of the controller.

Target and controller MUST be numeric. If the target is a floating point number and the controller an integer number, or vice versa, the sum is converted into the type of the target; converting from a floating point number to an integer selects its floor (the largest integer less than or equal to the floating point number).

```
interval<BASE> = (
  BASE => int             ; lower bound
  (BASE .plus 1) => int   ; upper bound
  ? (BASE .plus 2) => int ; tolerance
)

X = 0
Y = 3
rect = {
  interval<X>
  interval<Y>
}
```

Figure 2: Example: addition to a base value

The example in Figure 2 contains the generic definition of a group interval that gives a lower and an upper bound and optionally a tolerance. rect combines two of these groups into a map, one group for the X dimension and one for Y dimension.

3. Embedded ABNF

Many IETF protocols define allowable values for their text strings in ABNF [RFC5234] [RFC7405]. It is often desirable to define a text string type in CDDL by employing existing ABNF embedded into the CDDL specification. Without specific ABNF support in CDDL, that ABNF would usually need to be translated into a regular expression (if that is even possible).

ABNF is added to CDDL in the same way that regular expressions were added: by defining a .abnf control operator. The target is usually text or some restriction on it, the controller is the text of an ABNF specification.

There are several small issues, with solutions given here:

* ABNF can be used to define byte sequences as well as UTF-8 text strings interpreted as Unicode scalar sequences. This means this specification defines two control operators: .abnfb for ABNF denoting byte sequences and .abnf for denoting sequences of

Unicode scalar values (codepoint) represented as UTF-8 text strings. Both control operators can be applied to targets of either string type; the ABNF is applied to sequence of bytes in the string interpreting that as a sequence of bytes (.abnfb) or as a sequence of code points represented as an UTF-8 text string (.abnf). The controller string MUST be a text string.

* ABNF defines a list of rules, not a single expression (called "elements" in [RFC5234]). This is resolved by requiring the controller string to be one valid "element", followed by zero or more valid "rule" separated from the element by a newline; so the controller string can be built by preceding a piece of valid ABNF by an "element" that selects from that ABNF and a newline.

* For the same reason, ABNF requires newlines; specifying newlines in CDDL text strings is tedious (and leads to essentially unreadable ABNF). The workaround employs the .cat operator introduced in Section 2.1 and the syntax for text in byte strings. As is customary for ABNF, the syntax of ABNF itself (NOT the syntax expressed in ABNF!) is relaxed to allow a single linefeed as a newline:

```
CRLF = %x0A / %x0D.0A
```

* One set of rules provided in an ABNF specification is often used in multiple positions, in particular staples such as DIGIT and ALPHA. (Note that all rules referenced need to be defined in each ABNF operator controller string -- there is no implicit import of [RFC5234] Core ABNF or other rules.) The composition this calls for can be provided by the .cat operator.

These points are combined into an example in Figure 3, which uses ABNF from [RFC3339] to specify the CBOR tags defined in [I-D.ietf-cbor-date-tag].

```
; for draft-ietf-cbor-date-tag
Tag1004 = #6.1004(text .abnf full-date)
; for RFC 7049
Tag0 = #6.0(text .abnf date-time)

full-date = "full-date" .cat rfc3339
date-time = "date-time" .cat rfc3339

; Note the trick of idiomatically starting with a newline, separating
;   off the element in the .cat from the rule-list
rfc3339 = '
   date-fullyear   = 4DIGIT
   date-month      = 2DIGIT  ; 01-12
   date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                             ; month/year
   time-hour       = 2DIGIT  ; 00-23
   time-minute     = 2DIGIT  ; 00-59
   time-second     = 2DIGIT  ; 00-58, 00-59, 00-60 based on leap sec
                             ; rules
   time-secfrac    = "." 1*DIGIT
   time-numoffset  = ("+" / "-") time-hour ":" time-minute
   time-offset     = "Z" / time-numoffset

   partial-time    = time-hour ":" time-minute ":" time-second
                     [time-secfrac]
   full-date       = date-fullyear "-" date-month "-" date-mday
   full-time       = partial-time time-offset

   date-time       = full-date "T" full-time
' .cat rfc5234-core

rfc5234-core = '
         DIGIT          =  %x30-39 ; 0-9
; abbreviated here
'
```

Figure 3: Example: employing RFC 3339 ABNF for defining CBOR Tags

4. Features

Traditionally, the kind of validation enabled by languages such as CDDL provided a Boolean result: valid, or invalid.

In rapidly evolving environments, this is too simplistic. The data models described by a CDDL specification may continually be enhanced by additional features, and it would be useful even for a specification that does not yet describe a specific future feature to identify the extension point the feature can use, accepting such extensions while marking them as such.

The .feature control annotates the target as making use of the feature named by the controller. The latter will usually be a string. A tool that validates an instance against that specification may mark the instance as using a feature that is annotated by the specification.

Figure 4 shows what could be the definition of a person, with potential extensions beyond name and organization being marked further-person-extension. Extensions that are known at the time this definition is written can be collected into $$person-extensions. However, future extensions would be deemed invalid unless the wildcard at the end of the map is added. These extensions could then be specifically examined by a user or a tool that makes use of the validation result.

Leaving out the entire extension point would mean that instances that make use of an extension would be marked as whole-sale invalid, making the entire validation approach much less useful. Leaving the extension point in, but not marking its use as special, would render mistakes such as using the label organisation instead of organization invisible.

```
person = {
  ? name: text
  ? organization: text
  $$person-extensions
  * (text .feature "further-person-extension") => any
}

$$person-extensions //= (? bloodgroup: text)
```

Figure 4: Map extensibility with .feature

Figure 5 shows another example where .feature provides for type extensibility.

```
allowed-types = number / text / bool / null
              / [* number] / [* text] / [* bool]
              / (any .feature "allowed-type-extension")
```

Figure 5: Type extensibility with .feature

A CDDL tool may simply report the set of features being used; the control then only provides information to the process requesting the validation. One could also imagine a tool that takes arguments allowing the tool to accept certain features and reject others (enable/disable). The latter approach could for instance be used for a JSON/CBOR switch:

```
SenML-Record = {
; ...
  ? v => number
; ...
}
v = JC<"v", 2>
JC<J,C> = J .feature "json" / C .feature "cbor"
```

It remains to be seen if the enable/disable approach can lead to new idioms of using CDDL. The language currently has no way to enforce mutually exclusive use of features, as would be needed in this example.

5. IANA Considerations

This document requests IANA to register the contents of Table 2 into the CDDL Control Operators registry [IANA.cddl]:

Name	Reference
.cat	[RFCthis]
.plus	[RFCthis]
.abnf	[RFCthis]
.abnfb	[RFCthis]
.feature	[RFCthis]

Table 2

6. Implementation Status

An early implementation of the control operator .feature has been available in the CDDL tool since version 0.8.11. The validator warns about each feature being used and provides the set of target values used with the feature.

7. Security considerations

The security considerations of [RFC8610] apply.

8. References

8.1. Normative References

[IANA.cddl] IANA, "Concise Data Definition Language (CDDL)", , <http://www.iana.org/assignments/cddl>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC7405] Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, DOI 10.17487/RFC7405, December 2014, <https://www.rfc-editor.org/info/rfc7405>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

8.2. Informative References

[I-D.ietf-cbor-date-tag]
Jones, M., Nadalin, A., and J. Richter, "Concise Binary Object Representation (CBOR) Tags for Date", Work in Progress, Internet-Draft, draft-ietf-cbor-date-tag-06, 26 August 2020, <http://www.ietf.org/internet-drafts/draft-ietf-cbor-date-tag-06.txt>.

[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>.

Acknowledgements

Jim Schaad suggested several improvements. The .feature feature was developed out of a discussion with Henk Birkholz.

Author's Address

Carsten Bormann
Universität Bremen TZI
Postfach 330440
D-28359 Bremen
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org