                   Additional Control Operators for CDDL
                      draft-ietf-cbor-cddl-control-07

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 were
   not yet ready at the time RFC 8610 was completed: .plus, .cat and
   .det 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

   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 https://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 25 April 2022.

Copyright Notice

Table of Contents

1. Introduction

   The Concise Data Definition Language (CDDL), standardized in
   [RFC8610], provides "control operators" as its main language
   extension point (Section 3.8 of [RFC8610]).

   The present document defines a number of control operators that were
   not yet ready at the time RFC 8610 was completed:

```
+==========+=================================================+
| Name     | Purpose                                         |
+==========+=================================================+
| .plus    | Numeric addition                                |
+----------+-------------------------------------------------+
| .cat     | String Concatenation                            |
+----------+-------------------------------------------------+
| .det     | String Concatenation, pre-dedenting             |
+----------+-------------------------------------------------+
| .abnf    | ABNF in CDDL (text strings)                     |
+----------+-------------------------------------------------+
| .abnfb   | ABNF in CDDL (byte strings)                     |
+----------+-------------------------------------------------+
| .feature | Indicate name of feature used (extension point) |
+----------+-------------------------------------------------+
```

            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.
   "Tool" refers to tools along the lines of that described in
   Appendix F of [RFC8610]. Note also that the data model underlying
   CDDL provides for text strings as well as byte strings as two
   separate types, which are then collectively referred to as "strings".

   The term ABNF in this specification stands for the combination of
   [RFC5234] and [RFC7405], i.e., the ABNF control operators defined by
   this document allow use of the case-sensitive extensions defined in
   [RFC7405].

2. Computed Literals

   CDDL as defined in [RFC8610] does not have any mechanisms to compute
   literals. To cover a large part of the use cases, this specification
   adds three control operators: .plus for numeric addition, .cat for
   string concatenation, and .det for string concatenation with
   dedenting of both sides (target and controller).

For these operators, as with all control operators, targets and
controllers are types. The resulting type is therefore formally a
function of the elements of the cross-product of the two types. Not
all tools may be able to work with non-unique targets or controllers.

2.1. 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, i.e.,
rounding towards negative infinity).

```
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 1: Example: addition to a base value

The example in Figure 1 contains the generic definition of a CDDL
group interval that gives a lower and an upper bound and optionally a
tolerance. The parameter BASE allows the non-conflicting use of
multiple of these interval groups in one map, by assigning different
labels to the entries of the interval. rect combines two of these
interval groups into a map, one group for the X dimension (using 0,
1, and 2 as labels) and one for Y dimension (using 3, 4, and 5 as
labels).

2.2. 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. 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 2: Example: concatenation of text and byte string

The example in Figure 2 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.)

2.3. String Concatenation with Dedenting

Multi-line string literals for various applications, including
embedded ABNF (Section 3), need to be set flush left, at least
partially. Often, having some indentation in the source code for the
literal can promote readability, as in Figure 3.

```
oid = bytes .abnfb ("oid" .det cbor-tags-oid)
roid = bytes .abnfb ("roid" .det cbor-tags-oid)

cbor-tags-oid = '
  oid = 1*arc
  roid = *arc
  arc = [nlsb] %x00-7f
  nlsb = %x81-ff *%x80-ff
'
```

                  Figure 3: Example: dedenting concatenation

The control operator .det works like .cat, except that both arguments
(target and controller) are independently _dedented_ before the
concatenation takes place.

For the first rule in Figure 3, the result is equivalent to Figure 4.

```
oid = bytes .abnfb 'oid
oid = 1*arc
roid = *arc
arc = [nlsb] %x00-7f
nlsb = %x81-ff *%x80-ff
'
```

Figure 4: Dedenting example: result of first .det

For the purposes of this specification, we define dedenting as:

1. determining the smallest amount of left-most blank space (number
    of leading space characters) present in all the non-blank lines,
    and

2. removing exactly that number of leading space characters from
    each line. For blank (blank space only or empty) lines, there
    may be less (or no) leading space characters than this amount, in
    which case all leading space is removed.

(The name .det is a shortcut for "dedenting cat". The maybe more
obvious name .dedcat has not been chosen as it is longer and may
invoke unpleasant images.)

Occasionally, dedenting of only a single item is needed. This can be
achieved by using this operator with an empty string, e.g., "" .det
rhs or lhs .det "", which can in turn be combined with a .cat: in the
construct lhs .cat ("" .det rhs), only rhs is dedented.

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 2.2 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, and/or by .det if there
   is indentation to be disposed of.

These points are combined into an example in Figure 5, which uses
ABNF from [RFC3339] to specify one each of the CBOR tags defined in
[RFC8943] and [RFC8949].

```
   ; for RFC 8943
   Tag1004 = #6.1004(text .abnf full-date)
   ; for RFC 8949
   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 concatenations above 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
   ' .det rfc5234-core

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

   Figure 5: Example: employing RFC 3339 ABNF for defining CBOR Tags

4. Features

   Commonly, the kind of validation enabled by languages such as CDDL
   provides 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.

More specifically, the tool's diagnostic output might contain the
controller (right-hand side) as a feature name, and the target (left-
hand side) as a feature detail. However, in some cases, the target
has too much detail, and the specification might want to hint the
tool that more limited detail is appropriate. In this case, the
controller should be an array, with the first element being the
feature name (that would otherwise be the entire controller), and the
second element being the detail (usually another string), as
illustrated in Figure 6.

```
foo = {
  kind: bar / baz .feature (["foo-extensions", "bazify"])
}
bar = ...
baz = ... ; complex stuff that doesn't all need to be in the detail
```

              Figure 6: Providing explicit detail with .feature

Figure 7 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; the label (map key) actually used makes a fine
feature detail for the tool's diagnostic output.

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 7: Map extensibility with .feature

   Figure 8 shows another example where .feature provides for type
   extensibility.

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

                  Figure 8: 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, as illustrated in Figure 9, using SenML [RFC8428]
   as the example data model used with both JSON and CBOR.

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

                  Figure 9: Describing variants with .feature

   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 registry "CDDL Control Operators" of [IANA.cddl]:

```
+==========+===========+
| Name     | Reference |
+==========+===========+
| .plus    | [RFCthis] |
+----------+-----------+
| .cat     | [RFCthis] |
+----------+-----------+
| .det     | [RFCthis] |
+----------+-----------+
| .abnf    | [RFCthis] |
+----------+-----------+
| .abnfb   | [RFCthis] |
+----------+-----------+
| .feature | [RFCthis] |
+----------+-----------+
```

                         Table 2: New control
                            operators to be
                              registered

6. Implementation Status

   This section is to be removed before publishing as an RFC.

   An early implementation of the control operator .feature has been
   available in the CDDL tool described in Appendix F of [RFC8610] since
   version 0.8.11. The validator warns about each feature being used
   and provides the set of target values used with the feature. The
   other control operators defined in this specification are also
   implemented as of version 0.8.21 and 0.8.26 (double-handed .det).

   Andrew Weiss' [CDDL-RS] has an ongoing implementation of this draft
   which is feature-complete except for the ABNF and dedenting support
   (https://github.com/anweiss/cddl/pull/79
   (https://github.com/anweiss/cddl/pull/79)).

7. Security considerations

   The security considerations of [RFC8610] apply.

   While both [RFC5234] and [RFC7405] state that security is truly
   believed to be irrelevant to the respective document, the use of
   formal description techniques cannot only simplify, but sometimes
   also complicate a specification. This can lead to security problems
   in implementations and in the specification itself. As with CDDL
   itself, ABNF should be judiciously applied, and overly complex (or
   "cute") constructions should be avoided.

8. References

8.1. Normative References

   [IANA.cddl]
              IANA, "Concise Data Definition Language (CDDL)",
              <https://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

   [CDDL-RS] Weiss, A., "cddl-rs", n.d.,
              <https://github.com/anweiss/cddl>.

   [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>.

   [RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C.
              Bormann, "Sensor Measurement Lists (SenML)", RFC 8428,
              DOI 10.17487/RFC8428, August 2018,
              <https://www.rfc-editor.org/info/rfc8428>.

   [RFC8943]   Jones, M., Nadalin, A., and J. Richter, "Concise Binary
               Object Representation (CBOR) Tags for Date", RFC 8943,
               DOI 10.17487/RFC8943, November 2020,
               <https://www.rfc-editor.org/info/rfc8943>.

   [RFC8949]   Bormann, C. and P. Hoffman, "Concise Binary Object
               Representation (CBOR)", STD 94, RFC 8949,
               DOI 10.17487/RFC8949, December 2020,
               <https://www.rfc-editor.org/info/rfc8949>.

Acknowledgements

Author's Address

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

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