Workgroup: CBOR Working group
Internet-Draft:
draft-bormann-cbor-cddl-2-draft-01
Published: 6 March 2023
Intended Status: Informational
Expires: 7 September 2023
Authors: C. Bormann
Universität Bremen TZI

CDDL 2.0 — a draft plan

Abstract

The Concise Data Definition Language (CDDL) today is defined by RFC 8610 and RFC 9165. The latter (as well as some more application specific specifications such as RFC 9090) have used the extension point provided in RFC 8610, the control operator.

As CDDL is used in larger projects, feature requirements become known that cannot be easily mapped into this single extension point. Hence, there is a need for evolution of the base CDDL specification itself.

The present document provides a roadmap towards a "CDDL 2.0". It is based on draft-bormann-cbor-cddl-freezer, but is more selective in what potential features it takes up and more detailed in their discussion. It is intended to serve as a basis for prototypical implementations of CDDL 2.0. What specific documents spawn from the present one or whether this document is evolved into a single CDDL 2.0 specification.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-bormann-cbor-cddl-2-draft/.

Discussion of this document takes place on the cbor Working Group mailing list (mailto:cbor@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cbor/. Subscribe at https://www.ietf.org/mailman/listinfo/cbor/.

Source for this draft and an issue tracker can be found at https://github.com/cbor-wg/cddl-2.

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 7 September 2023.

Copyright Notice

Table of Contents

1. Introduction
2. Mending syntax deficits
   2.1. Empty data models
   2.2. Non-literal Tag Numbers
   2.3. Tag-oriented Literals
   2.4. Clarifications
      2.4.1. Err6527
      2.4.2. Err6543
3. Processing model: Beyond Validation
4. Module superstructure
   4.1. Compatibility
   4.2. Namespacing
   4.3. Cross-universe references
   4.4. The "module", "directives"
   4.5. Finding modules
   4.6. Initial Set of Directives
   4.7. Explicit selection of names
   4.8. Tool Support for Command-Line Control
   4.9. ABNF is a lot like CDDL
5. IANA Considerations

6. Security considerations
7. References
7.1. Normative References
7.2. Informative References
Appendix A. Fridge
A.1. Tag-oriented Literals
A.2. Cross-universe references
A.2.1. IANA references
Appendix B. A CDDL 2.0 Tool
Acknowledgements
Author's Address

1. Introduction

(Please see abstract.)

Note that the existing extension point can be exercised for new features in parallel to the work described here.

2. Mending syntax deficits

2.1. Empty data models

Proposal Status: complete
Compatibility: backward (not forward)

[RFC8610] requires a CDDL file to have at least one rule.

```
cddl = S 1*(rule S)
```

This makes sense when the file has to stand alone, as it needs to have at least one rule to provide an entry point (start rule).

With CDDL 2.0, CDDL files can also include directives (see Section 4.6), and these might be the source of all the rules that ultimately make up the module created by the file. The rule content has to be available for directive processing, making the requirement for at least one rule cumbersome.

Therefore, we extend the grammar as follows:

```
cddl = S *(rule S)
```

and make the existence of at least one rule a semantic constraint, to be fulfilled after processing of all directives.

2.2. Non-literal Tag Numbers

Proposal Status: complete
Compatibility: backward (not forward)

The CDDL 1.0 syntax for expressing tags in CDDL is (ABNF as in [RFC5234]):

```
type2 /= "#" "6" ["." uint] "(" S type S ")"
```

This means tag numbers can only be given as literal numbers (uints). Some specifications operate on ranges of tag numbers, e.g., [RFC9277] has a range of tag numbers 1668546817 (0x63740101) to 1668612095 (0x6374FFFF) to tag specific content formats. This can currently not be expressed in CDDL.

CDDL 2.0 extends this to

```
type2 /= "#" "6" ["." tag-number] "(" S type S ")"
tag-number = uint / ("<" type ">")
```

So the above range can be expressed in a CDDL fragment such as:

```
ct-tag<content> = #6.<ct-tag-number>(content)
ct-tag-number = 1668546817..1668612095
; or use 0x63740101..0x6374FFFF
```

Note that reuses the angle bracket syntax for generics; this reuse is innocuous as a generic parameter/argument only ever occurs after a rule name (id), while it occurs after . here. (Whether there is potential for human confusion can be debated; the above example deliberately uses generics as well.)

2.3. Tag-oriented Literals

Incomplete, see Appendix A.1.

2.4. Clarifications

Proposal Status: complete
Compatibility: errata fix (targets 1.0 and 2.0)

A number of errata reports have been made around some details of text string and byte string literal syntax: [Err6527] and [Err6543]. These need to be addressed by re-examining the details of these literal syntaxes. Also, [Err6526] needs to be applied (missing backslashes in text explaining backslash escaping).

2.4.1. Err6527

The ABNF used in [RFC8610] for the content of text string literals is rather permissive:

```
text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-7E / %x80-10FFFD / SESC
SESC = "\" (%x20-7E / %x80-10FFFD)
```

This allows almost any non-C0 character to be escaped by a backslash, but critically misses out on the \uXXXX and \uHHHH\uLLLL forms that JSON allows to specify characters in hex. Both can be solved by updating the SESC production to:

```
SESC = "\" ( %x22 / "/" / "\" /                 ; \" \/ \\
             %x62 / %x66 / %x6E / %x72 / %x74 / ; \b \f \n \r \t
             (%x75 hexchar) )                   ; \u
hexchar = non-surrogate / (high-surrogate "\" %x75 low-surrogate)
non-surrogate = ((DIGIT / "A"/"B"/"C" / "E"/"F") 3HEXDIG) /
                ("D" %x30-37 2HEXDIG )
high-surrogate = "D" ("8"/"9"/"A"/"B") 2HEXDIG
low-surrogate = "D" ("C"/"D"/"E"/"F") 2HEXDIG
```

Now that SESC is more restrictively formulated, this also requires an update to the BCHAR production used in the ABNF syntax for byte string literals:

```
bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = "h" / "b64"
```

The updated version explicit allows \', which is no longer allowed in the updated SESC:

```
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / "\'" / CRLF
```

2.4.2. Err6543

The ABNF used in [RFC8610] for the content of byte string literals lumps together byte strings notated as text with byte strings notated in base16 (hex) or base64 (but see also updated BCHAR production above):

```
bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
```

Errata report 6543 proposes to handle the two cases in separate productions (where, with an updated SESC, BCHAR obviously needs to be updated as above):

```
bytes = %x27 *BCHAR %x27
      / bsqual %x27 *QCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
QCHAR = DIGIT / ALPHA / "+" / "/" / "-" / "_" / "=" / WS
```

This potentially causes a subtle change, which is hidden in the WS production:

```
WS = SP / NL
SP = %x20
NL = COMMENT / CRLF
COMMENT = ";" *PCHAR CRLF
PCHAR = %x20-7E / %x80-10FFFD
CRLF = %x0A / %x0D.0A
```

This allows any non-C0 character in a comment, so this fragment becomes possible:

```
foo = h'
   43424F52 ; 'CBOR'
   0A       ; LF, but don't use CR!
'
```

The current text is not unambiguously saying whether the three apostrophes need to be escaped with a \ or not, as in:

```
foo = h'
   43424F52 ; \'CBOR\'
   0A       ; LF, but don\'t use CR!
'
```

... which would be supported by the existing ABNF in [RFC8610].

3. Processing model: Beyond Validation

Proposal Status: experiments with implementations ongoing
Compatibility: backwards compatible

The basic (implicit) processing model for CDDL 1.0 applies a CDDL data model to a data item and returns a Boolean that indicates whether the data item matches that model ("*validation*").

Section 4 of [RFC9165] extends this model with named "*features*". A validation can indicate which features were used. Validation could also be parameterized with information about what features are allowed to be used, enabling variants (see Section 4 of [RFC9165] and [useful] for examples).

The cddl tool (Appendix F of [RFC8610]) also supports experimental forms of "annotating" a validated data item with information about which rules were used to support validation, currently entirely based on the information that is in a standard CDDL 1.0 data model. This leads to a more general concept of "*annotation*", where the data model specification supports "annotating" the validated instance by optionally supplying information in the model. (The annotated result

is a special case of a "post-schema validation instance" [PSVI], here one where the data item itself is only augmented, not changed, by the process.)

Annotations could in turn provide input to further validation steps, as is often done with Schematron validation in Relax-NG; with an appropriate evaluation language this can be used for checking co-occurrence constraints (Section 5 of [I-D.draft-bormann-cbor-cddl-freezer]).

Finally, annotations are a first step to *transformation*, i.e., describing how a validated data item should be interpreted as a transformed data item by performing certain computations. This generally requires even more support from an evaluation language, simple transformations such as adding in default values may not need much support though.

At this time, existing experimental implementations do not lead to a clear choice for what processing model enhancements should be in CDDL 2.0. This document proposes to continue the experimentation and document good approaches.

4. Module superstructure

Proposal Status: collection of rough ideas with examples; initial subset implemented
Compatibility: bidirectional (both backward and forward)

Originally, CDDL was used for small data models that could be expressed in a few lines. As the size of data models that need to be expressed in CDDL has increased, the need to modularize and re-use components is increasing.

CDDL 1.0 has been designed with a crude form of composition: Concatenating a number of CDDL snippets creates a valid CDDL data model unless there is a name collision (identical redefinition is allowed to facilitate this approach). With larger models, managing the name space to avoid collisions becomes more pressing.

The knowledge which CDDL snippets need to be concatenated in order to obtain the desired data model lives entirely outside the CDDL snippets in CDDL 1.0. In CDDL 2.0, rules will be packaged as modules and referenced from other modules.

There needs to be some control of namespace pollution, as well as unambiguous referencing into evolving specifications ("versioning") and selection of alternatives (as was emulated with snippets in Section 11 of [RFC8428], although an alternative approach for expressing variants is demonstrated in [useful] based on Section 4 of [RFC9165]).

4.1. Compatibility

One approach to achieve the module structure that is friendly to existing environments that operate with CDDL 1.0 snippets and CDDL 1.0 implementations is to add a super-syntax (similar to the way pragmas are often added to a language), e.g., by carrying them in what is parsed as comments in CDDL 1.0.

This enables each module source file to be valid CDDL 1.0 (if missing some rule definitions to be imported).

4.2. Namespacing

A convention for mapping CDDL-internal names to external ones could be developed, possibly steered by some pragma-like constructs. External names would likely be URI-based, with some conventions as they are used in RDF or Curies. Internal names might look similar to XML QNames. Note that the identifier character set for CDDL deliberately includes $ and @, which could be used in such a convention.

Note that this convention should not pollute the actual contents of the model, where adding a simple prefix to rule names defined elsewhere may be all that is needed.

4.3. Cross-universe references

See Appendix A.2.

4.4. The "module", "directives"

A single CDDL file becomes a *module* by processing the (zero or more) *directives* in it.

The semantics of the module are independent of the module(s) using it, however, using a module may involve transforming its rule names into a new namespace.

Directives look like comments in CDDL 1.0, so they do not interfere with forward compatibility.

Lines starting with the prefix ;# are parsed as directives in CDDL 2.0.

4.5. Finding modules

For now, we assume that module names are filenames taken from one of several sources available to the CDDL 2.0 processor via the environment. This avoids the need to nail down pathnames or partial URIs into the CDDL files.

In the CDDL 2.0 Tool described in Appendix B, the set of sourced is determined from an environment value, CDDL_INCLUDE_PATH, which is modeled after usual command-line search paths. It is a colon-separated list of pathnames to directories, with one special feature: an empty element points to he tool's own collection. In the current version, this collection contains 20 fragments of extracted CDDL from published RFCs, using names such as rfc9052.

(Future versions might augment this with Web extractors and/or ways to extract CDDL modules from github and from Internet-Drafts.)

The default CDDL_INCLUDE_PATH is .: — i.e., files are found in the current directory and, if not found there, cddlc's collection.

4.6. Initial Set of Directives

Two groups of directives are defined at this point:

- include, which includes all the rules from a module (which includes the ones imported/included there, transitively), or specific explicitly selected rules

- import, which includes only those rules from the module that are referenced, implicitly or explicitly (see below), including the rules that are referenced from these rules, transitively.

The include function is more useful for composing a single model from parts controlled by one author, while the import function is more about treating a module as a library:

The way an import works is shown by this simple example:

```
$ cddlc -2tcddl -
start = COSE_Key
;# import rfc9052
```

This results in the following CDDL 1.0 specification:

```
start = COSE_Key
COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * label => values,
}
label = int / tstr
values = any
```

This is appropriate for using libraries that are well known to the imported. However, if it is not acceptable that the library can pollute the namespace of the importing module, the import directive can specify a namespace prefix:

```
$ cddlc -2tcddl -
start = cose.COSE_Key
;# import rfc9052 as cose
```

This results in the following CDDL 1.0 specification:

```
start = cose.COSE_Key
cose.COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * cose.label => cose.values,
}
cose.label = int / tstr
cose.values = any
```

Note how the imported names are prefixed with cose. as specified in the import directive, but CDDL prelude (Appendix D of [RFC8610]) names such as tstr and any are not.

4.7. Explicit selection of names

Both import and include directives can be augmented by an explicit mentioning of rule names.

Starting with include:

```
$ cddlc -2tcddl -
mydata = {* label => values}
;# include label, values from rfc9052
```

Only exactly the rules mentioned are included:

```
mydata = {* label => values}
label = int / tstr
values = any
```

The module from which rules are explicitly imported can be namespaced:

```
$ cddlc -2tcddl -
mydata = {* label => values}
;# include cose.label, cose.values from rfc9052 as cose
```

Again, only exactly the rules mentioned are included:

```
mydata = {* label => values}
cose.label = int / tstr
cose.values = any
```

Both examples would work exactly the same with import, as the included rules do not reference anything else from the included module.

An import however also draws in the transitive closure of the rules referenced:

```
$ cddlc -2tcddl -
mydata = {Fritz: cose.empty_or_serialized_map}
;# import cose.empty_or_serialized_map from rfc9052 as cose
```

The transitive closure of the rules mentioned is included:

```
mydata = {"Fritz" => cose.empty_or_serialized_map}
cose.empty_or_serialized_map = bstr .cbor cose.header_map / bstr .size 0
cose.header_map = {
  cose.Generic_Headers,
  * cose.label => cose.values,
}
cose.Generic_Headers = (
  ? 1 => int / tstr,
  ? 2 => [+ cose.label],
  ? 3 => tstr / int,
  ? 4 => bstr,
  ? (5 => bstr // 6 => bstr),
  )
cose.label = int / tstr
cose.values = any
```

The import statement can also request an alias for an imported name:

```
$ cddlc -2tcddl -
mydata = {Fritz: cose.empty_or_serialized_map}
;# import empty_or_serialized_map from rfc9052 as cose
```

Note how an additional rule provides an alias for empty_or_serialized_map that does not have the namespace prefix:

```
mydata = {"Fritz" => cose.empty_or_serialized_map}
empty_or_serialized_map = cose.empty_or_serialized_map
cose.empty_or_serialized_map = bstr .cbor cose.header_map / bstr .size 0
cose.header_map = {
  cose.Generic_Headers,
  * cose.label => cose.values,
}
cose.Generic_Headers = (
  ? 1 => int / tstr,
  ? 2 => [+ cose.label],
  ? 3 => tstr / int,
  ? 4 => bstr,
  ? (5 => bstr // 6 => bstr),
  )
cose.label = int / tstr
cose.values = any
```

4.8. Tool Support for Command-Line Control

A tool may provide a way to root the module tree from the command line:

```
$ cddlc -2tcddl -icose=rfc9052 -scose.COSE_Key
```

The command line argument -icose=rfc9052 is a shortcut for

```
;# import rfc9052 as cose
```

Together with the start rule name, cose.COSE_Key, this results in the following CDDL 1.0 specification:

```
$.start.$ = cose.COSE_Key
cose.COSE_Key = {
  1 => tstr / int,
  ? 2 => bstr,
  ? 3 => tstr / int,
  ? 4 => [+ tstr / int],
  ? 5 => bstr,
  * cose.label => cose.values,
}
cose.label = int / tstr
cose.values = any
```

In other words, the module had an empty CDDL file, which therefore was not provided (no - on the command line).

4.9. ABNF is a lot like CDDL

Many of the constructs defined here for CDDL also could be used with ABNF specifications. ABNF would definitely benefit from a standard way to import snippets from existing RFCs. Since CDDL contains ABNF support (Section 3 of [RFC9165]), it would be natural to make some of the functionality discussed in this section available for ABNF as well.

5. IANA Considerations

(Insert new registry for application specific literals here, if adopted.)

6. Security considerations

The security considerations of [RFC8610] apply.

7. References

7.1. Normative References

[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/rfc/rfc8610>.

[RFC9165] Bormann, C., "Additional Control Operators for the Concise Data Definition Language (CDDL)", RFC 9165, DOI 10.17487/RFC9165, December 2021, <https://www.rfc-editor.org/rfc/rfc9165>.

7.2. Informative References

[cddlc] "CDDL conversion utilities", n.d., <https://github.com/cabo/cddlc>.

[Err6526] "Errata Report 6526", RFC 8610, <https://www.rfc-editor.org/errata/eid6526>.

[Err6527] "Errata Report 6527", RFC 8610, <https://www.rfc-editor.org/errata/eid6527>.

[Err6543] "Errata Report 6543", RFC 8610, <https://www.rfc-editor.org/errata/eid6543>.

[I-D.bormann-cbor-edn-literals]
Bormann, C., "Application-Oriented Literals in CBOR Extended Diagnostic Notation", Work in Progress, Internet-Draft, draft-bormann-cbor-edn-literals-01, 24 October 2022, <https://datatracker.ietf.org/doc/html/draft-bormann-cbor-edn-literals-01>.

[I-D.draft-bormann-cbor-cddl-freezer]
Bormann, C., "A feature freezer for the Concise Data Definition Language (CDDL)", Work in Progress, Internet-Draft, draft-bormann-cbor-cddl-freezer-10, 24 October 2022, <https://datatracker.ietf.org/doc/html/draft-bormann-cbor-cddl-freezer-10>.

[PSVI] "Use Cases for XML Schema PSVI API", 24 June 2002, <https://www.w3.org/XML/2002/05/psvi-use-cases>.

[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/rfc/rfc5234>.

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

[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/rfc/rfc8428>.

[RFC9090] Bormann, C., "Concise Binary Object Representation (CBOR) Tags for Object Identifiers", RFC 9090, DOI 10.17487/RFC9090, July 2021, <https://www.rfc-editor.org/rfc/rfc9090>.

[RFC9277] Richardson, M. and C. Bormann, "On Stable Storage for Items in Concise Binary Object Representation (CBOR)", RFC 9277, DOI 10.17487/RFC9277, August 2022, <https://www.rfc-editor.org/rfc/rfc9277>.

[useful] "Useful CDDL", n.d., <https://github.com/cbor-wg/cddl/wiki/Useful-CDDL>.

Appendix A. Fridge

This appendix contains sections that may not make it to a 2.0, but might be part of a followup.

A.1. Tag-oriented Literals

Proposal Status: rough idea, porting from EDN
Compatibility: backward (not forward)

Some CBOR tags often would be most natural to use in a CDDL spec with a literal syntax that is tailored to their semantics instead of their serialization in CBOR. There is currently no way to add such syntaxes, no defined extension point either.

The proposal "Application-Oriented Literals in CBOR Extended Diagnostic Notation" [I-D.bormann-cbor-edn-literals] defines application-oriented literals, e.g., of the form

```
dt'2019-07-21T19:53Z'
```

for datetime items. With additional considerations for unambiguous syntax, a similar literal form could be included in CDDL.

This proposal opens a name space for the prefix that indicates an application specific literal. A registry could be provided to make this name space a genuine extension point. (This is currently the production bsqual in Appendix B of [RFC8610].)

The syntax provided in [I-D.bormann-cbor-edn-literals] does not enable the use of CDDL types — it has the same flaw that is being fixed for tag numbers in Section 2.2.

A.2. Cross-universe references

Often, a CDDL specification needs to import from specifications in a different language or platform.

A.2.1. IANA references

In many cases, CDDL specifications make use of values that are specified in IANA registries. The .iana control operator can be used to reference such a set of values.

The reference needs to be able to point to a draft, the registry of which has not been established yet, as well as to an established IANA registry.

An example of such a usage might be:

```
cose-algorithm = int .iana ["cose", "algorithms", "value"]
```

Unfortunately, the vocabulary employed in IANA registries has not been designed for machine references. In this case, the potential values would come from applying the XPath expression

```
//iana:registry[@id='algorithms']/iana:record/iana:value
```

to https://www.iana.org/assignments/cose/cose.xml, plus some filtering on the records returned that only leaves actual allocations. Additional functionality may be needed for filtering with respect to other columns of the registry record, e.g., <capabilities> in the case of this example.

Appendix B. A CDDL 2.0 Tool

This appendix is for information only.

A rough CDDL 2.0 tool is available [cddlc]. It can process CDDL 2.0 models into CDDL 1.0 models that can then be processed by the CDDL tool described in Appendix F of [RFC8610].

A typical command line involving both tools might be:

```
cddlc -2 -tcddl mytestfile.cddl | cddl - gp 10
```

Install on a system with a modern Ruby (Ruby version ≥ 3.0) via:

```
gem install cddlc
```

The present document assumes the use of cddlc version 0.1.5.

Acknowledgements

TBD

Author's Address

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

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