Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 30, 2018

H. Birkholz
Fraunhofer SIT
C. Vigano
Universitaet Bremen
C. Bormann
Universitaet Bremen TZI

February 26, 2018

Concise data definition language (CDDL): a notational convention to express CBOR data structures
draft-ietf-cbor-cddl-02

Abstract

This document proposes a notational convention to express CBOR data structures (RFC 7049). Its main goal is to provide an easy and unambiguous way to express structures for protocol messages and data formats that use CBOR.

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 August 30, 2018.

Table of Contents

1. Introduction 3
1.1. Requirements notation 4
1.2. Terminology 4
2. The Style of Data Structure Specification 4
2.1. Groups and Composition in CDDL 6
2.1.1. Usage 8
2.1.2. Syntax 8
2.2. Types 9
2.2.1. Values 9
2.2.2. Choices 9
2.2.3. Representation Types 11
2.2.4. Root type 11
3. Syntax 12
3.1. General conventions 12
3.2. Occurrence 13
3.3. Predefined names for types 14
3.4. Arrays 15
3.5. Maps 15
3.5.1. Structs 16
3.5.2. Tables 19
3.5.3. Cuts in Maps 19
3.6. Tags 20
3.7. Unwrapping 21
3.8. Controls 22
3.8.1. Control operator .size 22
3.8.2. Control operator .bits 23
3.8.3. Control operator .regexp 24
3.8.4. Control operators .cbor and .cborseq 25
3.8.5. Control operators .within and .and 25
3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default 26
3.9. Socket/Plug 27
3.10. Generics 28
3.11. Operator Precedence 28
4. Making Use of CDDL 30
4.1. As a guide to a human user 30
4.2. For automated checking of CBOR data structure 30
4.3. For data analysis tools 31
5. Security considerations 31
6. IANA considerations 31
7. References 32
7.1. Normative References 32

7.2. Informative References 32
Appendix A. (Not used.) 33
Appendix B. ABNF grammar 33
Appendix C. Matching rules 36
Appendix D. (Not used.) 40
Appendix E. Standard Prelude 40
E.1. Use with JSON 42
Appendix F. The CDDL tool 44
Appendix G. Extended Diagnostic Notation 44
G.1. White space in byte string notation 45
G.2. Text in byte string notation 45
G.3. Embedded CBOR and CBOR sequences in byte strings 45
G.4. Concatenated Strings 46
G.5. Hexadecimal, octal, and binary numbers 46
G.6. Comments 47
Appendix H. Examples 47
H.1. RFC 7071 48
H.1.1. Examples from JSON Content Rules 52
Acknowledgements 54
Authors' Addresses 55

1. Introduction

In this document, a notational convention to express CBOR [RFC7049] data structures is defined.

The main goal for the convention is to provide a unified notation that can be used when defining protocols that use CBOR. We term the convention "Concise data definition language", or CDDL.

The CBOR notational convention has the following goals:

(G1) Provide an unambiguous description of the overall structure of a CBOR data structure.

(G2) Flexibility to express the freedoms of choice in the CBOR data format.

(G3) Possibility to restrict format choices where appropriate [_format].

(G4) Able to express common CBOR datatypes and structures.

(G5) Human and machine readable and processable.

(G6) Automatic checking of data format compliance.

(G7) Extraction of specific elements from CBOR data for further processing.

Not an explicit goal per se, but a convenient side effect of the JSON generic data model being a subset of the CBOR generic data model, is the fact that CDDL can also be used for describing JSON data structures (see Appendix E.1).

This document has the following structure:

The syntax of CDDL is defined in Section 3. Examples of CDDL and related CBOR data items ("instances") are defined in Appendix H. Section 4 discusses usage of CDDL. Examples are provided early in the text to better illustrate concept definitions. A formal definition of CDDL using ABNF grammar is provided in Appendix B. Finally, a prelude of standard CDDL definitions available in every CBOR specification is listed in Appendix E.

1.1. Requirements notation

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 RFC 2119, BCP 14 [RFC2119].

1.2. Terminology

New terms are introduced in _cursive_. CDDL text in the running text is in "typewriter".

2. The Style of Data Structure Specification

CDDL focuses on styles of specification that are in use in the community employing the data model as pioneered by JSON and now refined in CBOR.

There are a number of more or less atomic elements of a CBOR data model, such as numbers, simple values (false, true, nil), text and byte strings; CDDL does not focus on specifying their structure. CDDL of course also allows adding a CBOR tag to a data item.

The more important components of a data structure definition language are the data types used for composition: arrays and maps in CBOR (called arrays and objects in JSON). While these are only two representation formats, they are used to specify four loosely distinguishable styles of composition:

- A _vector_, an array of elements that are mostly of the same semantics. The set of signatures associated with a signed data item is a typical application of a vector.

- A _record_, an array the elements of which have different, positionally defined semantics, as detailed in the data structure definition. A 2D point, specified as an array of an x coordinate (which comes first) and a y coordinate (coming second) is an example of a record, as is the pair of exponent (first) and mantissa (second) in a CBOR decimal fraction.

- A _table_, a map from a domain of map keys to a domain of map values, that are mostly of the same semantics. A set of language tags, each mapped to a text string translated to that specific language, is an example of a table. The key domain is usually not limited to a specific set by the specification, but open for the application, e.g., in a table mapping IP addresses to MAC addresses, the specification does not attempt to foresee all possible IP addresses.

- A _struct_, a map from a domain of map keys as defined by the specification to a domain of map values the semantics of each of which is bound to a specific map key. This is what many people have in mind when they think about JSON objects; CBOR adds the ability to use map keys that are not just text strings. Structs can be used to solve similar problems as records; the use of explicit map keys facilitates optionality and extensibility.

Two important concepts provide the foundation for CDDL:

1. Instead of defining all four types of composition in CDDL separately, or even defining one kind for arrays (vectors and records) and one kind for maps (tables and structs), there is only one kind of composition in CDDL: the _group_ (Section 2.1).

2. The other important concept is that of a _type_. The entire CDDL specification defines a type (the one defined by its first _rule_), which formally is the set of CBOR data items that are acceptable as "instances" for this specification. CDDL predefines a number of basic types such as "uint" (unsigned integer) or "tstr" (text string), often making use of a simple formal notation for CBOR data items. Each value that can be expressed as a CBOR data item also is a type in its own right, e.g. "1". A type can be built as a _choice_ of other types, e.g., an "int" is either a "uint" or a "nint" (negative integer). Finally, a type can be built as an array or a map from a group.

The rest of this section introduces a number of basic concepts of CDDL, and section Section 3 defines additional syntax. Appendix C gives a concise summary of the semantics of CDDL.

2.1. Groups and Composition in CDDL

CDDL Groups are lists of name/value pairs (group _entries_).

In an array context, only the value of the entry is represented; the name is annotation only (and can be left off if not needed). In a map context, the names become the map keys ("member keys").

In an array context, the sequence of elements in the group is important, as it is the information that allows associating actual array elements with entries in the group. In a map context, the sequence of entries in a group is not relevant (but there is still a need to write down group entries in a sequence).

A simple example of using a group right in a map definition is:

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}
```

Figure 1: Using a group in a map

The three entries of the group are written between the curly braces that create the map: Here, "age", "name", and "employer" are the names that turn into the map key text strings, and "int" and "tstr" (text string) are the types of the map values under these keys.

A group by itself (without creating a map around it) can be placed in (round) parentheses, and given a name by using it in a rule:

```
pii = (
  age: int,
  name: tstr,
  employer: tstr,
)
```

Figure 2: A basic group

This separate, named group definition allows us to rephrase Figure 1 as:

```
person = {
  pii
}
```

Figure 3: Using a group by name

Note that the (curly) braces signify the creation of a map; the groups themselves are neutral as to whether they will be used in a map or an array.

As shown in Figure 1, the parentheses for groups are optional when there is some other set of brackets present. Note that they can still be used, leading to the not so realistic, but perfectly valid example:

```
person = {(
  age: int,
  name: tstr,
  employer: tstr,
 )}
```

Groups can be used to factor out common parts of structs, e.g., instead of writing copy/paste style specifications such as in Figure 4, one can factor out the common subgroup, choose a name for it, and write only the specific parts into the individual maps (Figure 5).

```
person = {
  age: int,
  name: tstr,
  employer: tstr,
}

dog = {
  age: int,
  name: tstr,
  leash-length: float,
}
```

Figure 4: Maps with copy/paste

```
person = {
  identity,
  employer: tstr,
}

dog = {
  identity,
  leash-length: float,
}

identity = (
  age: int,
  name: tstr,
)
```

Figure 5: Using a group for factorization

Note that the lists inside the braces in the above definitions constitute (anonymous) groups, while "identity" is a named group.

2.1.1. Usage

Groups are the instrument used in composing data structures with CDDL. It is a matter of style in defining those structures whether to define groups (anonymously) right in their contexts or whether to define them in a separate rule and to reference them with their respective name (possibly more than once).

With this, one is allowed to define all small parts of their data structures and compose bigger protocol units with those or to have only one big protocol data unit that has all definitions ad hoc where needed.

2.1.2. Syntax

The composition syntax intends to be concise and easy to read:

- The start of a group can be marked by '('
- The end of a group can be marked by ')'
- Definitions of entries inside of a group are noted as follows: _keytype => valuetype,_ (read "keytype maps to valuetype"). The comma is actually optional (not just in the final entry), but it is considered good style to set it. The double arrow can be replaced by a colon in the common case of directly using a text string or integer literal as a key (see Section 3.5.1).

An entry consists of a _keytype_ and a _valuetype_:

- _keytype_ is either an atom used as the actual key or a type in general. The latter case may be needed when using groups in a table context, where the actual keys are of lesser importance than the key types, e.g in contexts verifying incoming data.

- _valuetype_ is a type, which could be derived from the major types defined in [RFC7049], could be a convenience valuetype defined in this document (Appendix E) or the name of a type defined in the specification.

A group definition can also contain choices between groups, see Section 2.2.2.

2.2. Types

2.2.1. Values

Values such as numbers and strings can be used in place of a type. (For instance, this is a very common thing to do for a keytype, common enough that CDDL provides additional convenience syntax for this.)

2.2.2. Choices

Many places that allow a type also allow a choice between types, delimited by a "/" (slash). The entire choice construct can be put into parentheses if this is required to make the construction unambiguous (please see Appendix B for the details).

Choices of values can be used to express enumerations:

```
attire = "bow tie" / "necktie" / "Internet attire"
protocol = 6 / 17
```

Similarly as for types, CDDL also allows choices between groups, delimited by a "//" (double slash).

```
address = { delivery }

delivery = (
street: tstr, ? number: uint, city //
po-box: uint, city //
per-pickup: true )

city = (
name: tstr, zip-code: uint
)
```

Both for type choices and for group choices, additional alternatives can be added to a rule later in separate rules by using "/=" and "//=", respectively, instead of "=":

```
attire /= "swimwear"

delivery //= (
lat: float, long: float, drone-type: tstr
)
```

It is not an error if a name is first used with a "/=" or "//=" (there is no need to "create it" with "=").

2.2.2.1. Ranges

Instead of naming all the values that make up a choice, CDDL allows building a _range_ out of two values that are in an ordering relationship. A range can be inclusive of both ends given (denoted by joining two values by ".."), or include the first and exclude the second (denoted by instead using "...").

```
device-address = byte
max-byte = 255
byte = 0..max-byte ; inclusive range
first-non-byte = 256
byte1 = 0...first-non-byte ; byte1 is equivalent to byte
```

CDDL currently only allows ranges between numbers [_range].

2.2.2.2. Turning a group into a choice

Some choices are built out of large numbers of values, often integers, each of which is best given a semantic name in the specification. Instead of naming each of these integers and then accumulating these into a choice, CDDL allows building a choice from a group by prefixing it with a "&" character:

```
terminal-color = &basecolors
basecolors = (
  black: 0, red: 1,  green: 2,  yellow: 3,
  blue: 4,  magenta: 5,  cyan: 6,  white: 7,
)
extended-color = &(
  basecolors,
  orange: 8,  pink: 9,  purple: 10,  brown: 11,
)
```

As with the use of groups in arrays (Section 3.4), the membernames have only documentary value (in particular, they might be used by a tool when displaying integers that are taken from that choice).

2.2.3. Representation Types

CDDL allows the specification of a data item type by referring to the CBOR representation (major and minor numbers). How this is used should be evident from the prelude (Appendix E).

It may be necessary to make use of representation types outside the prelude, e.g., a specification could start by making use of an existing tag in a more specific way, or define a new tag not defined in the prelude:

```
my_breakfast = #6.55799(breakfast)   ; cbor-any is too general!
breakfast = cereal / porridge
cereal = #6.998(tstr)
porridge = #6.999([liquid, solid])
liquid = milk / water
milk = 0
water = 1
solid = tstr
```

2.2.4. Root type

There is no special syntax to identify the root of a CDDL data structure definition: that role is simply taken by the first rule defined in the file.

This is motivated by the usual top-down approach for defining data structures, decomposing a big data structure unit into smaller parts; however, except for the root type, there is no need to strictly follow this sequence.

(Note that there is no way to use a group as a root - it must be a type. Using a group as the root might be employed as a way to specify a CBOR sequence in a future version of this specification;

this would act as if that group is used in an array and the data items in that fictional array form the members of the CBOR sequence.)

3. Syntax

In this section, the overall syntax of CDDL is shown, alongside some examples just illustrating syntax. (The definition will not attempt to be overly formal; refer to Appendix B for the details.)

3.1. General conventions

The basic syntax is inspired by ABNF [RFC5234], with

- rules, whether they define groups or types, are defined with a name, followed by an equals sign "=" and the actual definition according to the respective syntactic rules of that definition.

- A name can consist of any of the characters from the set {'A', ..., 'Z', 'a', ..., 'z', '0', ..., '9', '_', '-', '@', '.', '$'}, starting with an alphabetic character (including '@', '_', '$') and ending in one or a digit.

  * Names are case sensitive.

  * It is preferred style to start a name with a lower case letter.

  * The hyphen is preferred over the underscore (except in a "bareword" (Section 3.5.1), where the semantics may actually require an underscore).

  * The period may be useful for larger specifications, to express some module structure (as in "tcp.throughput" vs. "udp.throughput").

  * A number of names are predefined in the CDDL prelude, as listed in Appendix E.

  * Rule names (types or groups) do not appear in the actual CBOR encoding, but names used as "barewords" in member keys do.

- Comments are started by a ';' (semicolon) character and finish at the end of a line (LF or CRLF).

- outside strings, whitespace (spaces, newlines, and comments) is used to separate syntactic elements for readability (and to separate identifiers or numbers that follow each other); it is otherwise completely optional.

- Hexadecimal numbers are preceded by '0x' (without quotes, lower case x), and are case insensitive. Similarly, binary numbers are preceded by '0b'.

- Text strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in section 7 of [RFC8259]. (ABNF users may want to note that there is no support in CDDL for the concept of case insensitivity in text strings; if necessary, regular expressions can be used (Section 3.8.3).)

- Byte strings are enclosed by single quotation "'" characters and may be prefixed by "h" or "b64". If unprefixed, the string is interpreted as with a text string, except that single quotes must be escaped and that the UTF-8 bytes resulting are marked as a byte string (major type 2). If prefixed as "h" or "b64", the string is interpreted as a sequence of hex digits or a base64(url) string, respectively (as with the diagnostic notation in section 6 of [RFC7049]; cf. Appendix G.2); any white space present within the string (including comments) is ignored in the prefixed case. [_strings]

- CDDL uses UTF-8 [RFC3629] for its encoding.

Example:

```
; This is a comment
person = { g }

g = (
  "name": tstr,
  age: int,  ; "age" is a bareword
)
```

3.2. Occurrence

An optional _occurrence_ indicator can be given in front of a group entry. It is either one of the characters '?' (optional), '*' (zero or more), or '+' (one or more), or is of the form n*m, where n and m are optional unsigned integers and n is the lower limit (default 0) and m is the upper limit (default no limit) of occurrences.

If no occurrence indicator is specified, the group entry is to occur exactly once (as if 1*1 were specified).

Note that CDDL, outside any directives/annotations that could possibly be defined, does not make any prescription as to whether arrays or maps use the definite length or indefinite length encoding. I.e., there is no correlation between leaving the size of an array

"open" in the spec and the fact that it is then interchanged with definite or indefinite length.

Please also note that CDDL can describe flexibility that the data model of the target representation does not have. This is rather obvious for JSON, but also is relevant for CBOR:

```
apartment = {
  kitchen: size,
  * bedroom: size,
}
size = float ; in m2
```

The previous specification does not mean that CBOR is changed to allow to use the key "bedroom" more than once. In other words, due to the restrictions imposed by the data model, the third line pretty much turns into:

```
  ? bedroom: size,
```

(Occurrence indicators beyond one still are useful in maps for groups that allow a variety of keys.)

3.3. Predefined names for types

CDDL predefines a number of names. This subsection summarizes these names, but please see Appendix E for the exact definitions.

The following keywords for primitive datatypes are defined:

"bool" Boolean value (major type 7, additional information 20 or 21).

"uint" An unsigned integer (major type 0).

"nint" A negative integer (major type 1).

"int" An unsigned integer or a negative integer.

"float16" A number representable as an IEEE 754 half-precision float (major type 7, additional information 25).

"float32" A number representable as an IEEE 754 single-precision float (major type 7, additional information 26).

"float64" A number representable as an IEEE 754 double-precision float (major type 7, additional information 27).

"float" One of float16, float32, or float64.

"bstr" or "bytes" A byte string (major type 2).

"tstr" or "text" Text string (major type 3)

(Note that there are no predefined names for arrays or maps; these are defined with the syntax given below.)

In addition, a number of types are defined in the prelude that are associated with CBOR tags, such as "tdate", "bigint", "regexp" etc.

3.4. Arrays

Array definitions surround a group with square brackets.

For each entry, an occurrence indicator as specified in Section 3.2 is permitted.

For example:

```
unlimited-people = [* person]
one-or-two-people = [1*2 person]
at-least-two-people = [2* person]
person = (
    name: tstr,
    age: uint,
)
```

The group "person" is defined in such a way that repeating it in the array each time generates alternating names and ages, so these are four valid values for a data item of type "unlimited-people":

```
["roundlet", 1047, "psychurgy", 2204, "extrarhythmical", 2231]
[]
["aluminize", 212, "climograph", 4124]
["penintime", 1513, "endocarditis", 4084, "impermeator", 1669,
 "coextension", 865]
```

3.5. Maps

The syntax for specifying maps merits special attention, as well as a number of optimizations and conveniences, as it is likely to be the focal point of many specifications employing CDDL. While the syntax does not strictly distinguish struct and table usage of maps, it caters specifically to each of them.

But first, let's reiterate a feature of CBOR that it has inherited from JSON: The key/value pairs in CBOR maps have no fixed ordering. (One could imagine situations where fixing the ordering may be of use. For example, a decoder could look for values related with integer keys 1, 3 and 7. If the order were fixed and the decoder encounters the key 4 without having encountered key 3, it could conclude that key 3 is not available without doing more complicated bookkeeping. Unfortunately, neither JSON nor CBOR support this, so no attempt was made to support this in CDDL either.)

3.5.1. Structs

The "struct" usage of maps is similar to the way JSON objects are used in many JSON applications.

A map is defined in the same way as defining an array (see Section 3.4), except for using curly braces "{}" instead of square brackets "[]".

An occurrence indicator as specified in Section 3.2 is permitted for each group entry.

The following is an example of a structure:

```
Geography = [
  city           : tstr,
  gpsCoordinates : GpsCoordinates,
]

GpsCoordinates = {
  longitude      : uint,            ; multiplied by 10^7
  latitude       : uint,            ; multiplied by 10^7
}
```

When encoding, the Geography structure is encoded using a CBOR array with two entries (the keys for the group entries are ignored), whereas the GpsCoordinates are encoded as a CBOR map with two key/value pairs.

Types used in a structure can be defined in separate rules or just in place (potentially placed inside parentheses, such as for choices). E.g.:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
}
```

where "located-samples" is the datatype to be used when referring to the struct, and "sample-point" and "samples" are the keys to be used. This is actually a complete example: an identifier that is followed by a colon can be directly used as the text string for a member key (we speak of a "bareword" member key), as can a double-quoted string or a number. (When other types, in particular multi-valued ones, are used as keytypes, they are followed by a double arrow, see below.)

If a text string key does not match the syntax for an identifier (or if the specifier just happens to prefer using double quotes), the text string syntax can also be used in the member key position, followed by a colon. The above example could therefore have been written with quoted strings in the member key positions. More generally, all the types defined can be used in a keytype position by following them with a double arrow. A string also is a (single-valued) type, so another form for this example is:

```
located-samples = {
  "sample-point" => int,
  "samples" => [+ float],
}
```

See Section 3.5.3 below for how the colon shortcut described here also adds some implied semantics.

A better way to demonstrate the double-arrow use may be:

```
located-samples = {
  sample-point: int,
  samples: [+ float],
  * equipment-type => equipment-tolerances,
}
equipment-type = [name: tstr, manufacturer: tstr]
equipment-tolerances = [+ [float, float]]
```

The example below defines a struct with optional entries: display name (as a text string), the name components first name and family name (as a map of text strings), and age information (as an unsigned integer).

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Note that the group definition for NameComponents does not generate another map; instead, all four keys are directly in the struct built by PersonalData.

In this example, all key/value pairs are optional from the perspective of CDDL. With no occurrence indicator, an entry is mandatory.

If the addition of more entries not specified by the current specification is desired, one can add this possibility explicitly:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * tstr => any
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)
```

Figure 6: Personal Data: Example for extensibility

The cddl tool (Appendix F) generated as one acceptable instance for this specification:

```
{"familyName": "agust", "antiforeignism": "pretzel",
 "springbuck": "illuminatingly", "exuviae": "ephemeris",
 "kilometrage": "frogfish"}
```

(See Section 3.9 for one way to explicitly identify an extension point.)

3.5.2. Tables

A table can be specified by defining a map with entries where the keytype is not single-valued, e.g.:

```
square-roots = {* x => y}
x = int
y = float
```

Here, the key in each key/value pair has datatype x (defined as int), and the value has datatype y (defined as float).

If the specification does not need to restrict one of x or y (i.e., the application is free to choose per entry), it can be replaced by the predefined name "any".

As another example, the following could be used as a conversion table converting from an integer or float to a string:

```
tostring = {* mynumber => tstr}
mynumber = int / float
```

3.5.3. Cuts in Maps

The extensibility idiom discussed above for structs has one problem:

```
extensible-map-example = {
  ? "optional-key" => int,
  * tstr => any
}
```

In this example, there is one optional key "optional-key", which, when present, maps to an integer. There is also a wild card for any future additions.

Unfortunately, the data item

```
{ "optional-key": "nonsense" }
```

does match this specification: While the first entry of the group does not match, the second one (the wildcard) does. This may be very well desirable (e.g., if a future extension is to be allowed to extend the type of "optional-key"), but in many cases isn't.

In anticipation of a more general potential feature called "cuts", CDDL allows inserting a cut "^" into the definition of the map entry:

```
extensible-map-example = {
  ? "optional-key" ^ => int,
  * tstr => any
}
```

A cut in this position means that once the map key matches the entry carrying the cut, other potential matches for the key that occur in later entries in the group of the map are no longer allowed. (This rule applies independent of whether the value matches, too.) So the example above no longer matches the version modified with a cut.

Since the desire for this kind of exclusive matching is so frequent, the ":" shortcut is actually defined to include the cut semantics. So the preceding example (including the cut) can be written more simply as:

```
extensible-map-example = {
  ? "optional-key": int,
  * tstr => any
}
```

or even shorter, using a bareword for the key:

```
extensible-map-example = {
  ? optional-key: int,
  * tstr => any
}
```

3.6. Tags

A type can make use of a CBOR tag (major type 6) by using the representation type notation, giving #6.nnn(type) where nnn is an unsigned integer giving the tag number and "type" is the type of the data item being tagged.

For example, the following line from the CDDL prelude (Appendix E) defines "biguint" as a type name for a positive bignum N:

```
biguint = #6.2(bstr)
```

The tags defined by [RFC7049] are included in the prelude. Additional tags since registered need to be added to a CDDL specification as needed; e.g., a binary UUID tag could be referenced as "buuid" in a specification after defining

```
buuid = #6.37(bstr)
```

In the following example, usage of the tag 32 for URIs is optional:

```
my_uri = #6.32(tstr) / tstr
```

3.7. Unwrapping

The group that is used to define a map or an array can often be reused in the definition of another map or array. Similarly, a type defined as a tag carries an internal data item that one would like to refer to. In these cases, it is expedient to simply use the name of the map, array, or tag type as a handle for the group or type defined inside it.

The "unwrap" operator (written by preceding a name by a tilde character "~") can be used to strip the type defined for a name by one layer, exposing the underlying group (for maps and arrays) or type (for tags).

For example, an application might want to define a basic and an advanced header. Without unwrapping, this might be done as follows:

```
basic-header-group = (
  field1: int,
  field2: text,
)

basic-header = { basic-header-group }

advanced-header = {
  basic-header-group,
  field3: bytes,
  field4: number, ; as in the tagged type "time"
}
```

Unwrapping simplifies this to:

```
basic-header = {
  field1: int,
  field2: text,
}

advanced-header = {
  ~basic-header,
  field3: bytes,
  field4: ~time,
}
```

(Note that leaving out the first unwrap operator in the latter example would lead to nesting the basic-header in its own map inside the advanced-header, while, with the unwrapped basic-header, the

definition of the group inside basic-header is essentially repeated inside advanced-header, leading to a single map. This can be used for various applications often solved by inheritance in programming languages. The effect of unwrapping can also be described as "threading in" the group or type inside the referenced type, which suggested the thread-like "~" character.)

3.8. Controls

A _control_ allows to relate a _target_ type with a _controller_ type via a _control operator_.

The syntax for a control type is "target .control-operator controller", where control operators are special identifiers prefixed by a dot. (Note that _target_ or _controller_ might need to be parenthesized.)

A number of control operators are defined at his point. Note that the CDDL tool does not currently support combining multiple controls on a single target.

3.8.1. Control operator .size

A ".size" control controls the size of the target in bytes by the control type. Examples:

```
full-address = [[+ label], ip4, ip6]
ip4 = bstr .size 4
ip6 = bstr .size 16
label = bstr .size (1..63)
```

Figure 7: Control for size in bytes

When applied to an unsigned integer, the ".size" control restricts the range of that integer by giving a maximum number of bytes that should be needed in a computer representation of that unsigned integer. In other words, "uint .size N" is equivalent to "0...BYTES_N", where BYTES_N == 256**N.

```
audio_sample = uint .size 3 ; 24-bit, equivalent to 0..16777215
```

Figure 8: Control for integer size in bytes

Note that, as with value restrictions in CDDL, this control is not a representation constraint; a number that fits into fewer bytes can still be represented in that form, and an inefficient implementation could use a longer form (unless that is restricted by some format

constraints outside of CDDL, such as the rules in Section 3.9 of [RFC7049]).

3.8.2. Control operator .bits

A ".bits" control on a byte string indicates that, in the target, only the bits numbered by a number in the control type are allowed to be set. (Bits are counted the usual way, bit number "n" being set in "str" meaning that "(str[n >> 3] & (1 << (n & 7))) != 0".) [_bitsendian]

Similarly, a ".bits" control on an unsigned integer "i" indicates that for all unsigned integers "n" where "(i & (1 << n)) != 0", "n" must be in the control type.

```
tcpflagbytes = bstr .bits flags
flags = &(
  fin: 8,
  syn: 9,
  rst: 10,
  psh: 11,
  ack: 12,
  urg: 13,
  ece: 14,
  cwr: 15,
  ns: 0,
) / (4..7) ; data offset bits

rwxbits = uint .bits rwx
rwx = &(r: 2, w: 1, x: 0)
```

Figure 9: Control for what bits can be set

The CDDL tool generates the following ten example instances for "tcpflagbytes":

```
h'906d' h'01fc' h'8145' h'01b7' h'013d' h'409f' h'018e' h'c05f'
h'01fa' h'01fe'
```

These examples do not illustrate that the above CDDL specification does not explicitly specify a size of two bytes: A valid all clear instance of flag bytes could be "h''" or "h'00'" or even "h'000000'" as well.

3.8.3. Control operator .regexp

A ".regexp" control indicates that the text string given as a target needs to match the XSD regular expression given as a value in the control type. XSD regular expressions are defined in Appendix F of [W3C.REC-xmlschema-2-20041028].

```
nai = tstr .regexp "[A-Za-z0-9]+@[A-Za-z0-9]+(\\.[A-Za-z0-9]+)+"
```

Figure 10: Control with an XSD regexp

The CDDL tool proposes:

"N1@CH57HF.4Znqe0.dYJRN.igjf"

3.8.3.1. Usage considerations

Note that XSD regular expressions do not support the usual \x or \u escapes for hexadecimal expression of bytes or unicode code points. However, in CDDL the XSD regular expressions are contained in text strings, the literal notation for which provides \u escapes; this should suffice for most applications that use regular expressions for text strings. (Note that this also means that there is one level of string escaping before the XSD escaping rules are applied.)

XSD regular expressions support character class subtraction, a feature often not found in regular expression libraries; specification writers may want to use this feature sparingly. Similar considerations apply to Unicode character classes; where these are used, the specification SHOULD identify which Unicode versions are addressed.

Other surprises for infrequent users of XSD regular expressions may include:

- No direct support for case insensitivity. While case insensitivity has gone mostly out of fashion in protocol design, it is sometimes needed and then needs to be expressed manually as in "[Cc][Aa][Ss][Ee]".

- The support for popular character classes such as \w and \d is based on Unicode character properties, which is often not what is desired in an ASCII-based protocol and thus might lead to surprises. (\s and \S do have their more conventional meanings, and "." matches any character but the line ending characters \r or \n.)

3.8.3.2. Discussion

There are many flavors of regular expression in use in the programming community. For instance, perl-compatible regular expressions (PCRE) are widely used and probably are more useful than XSD regular expressions. However, there is no normative reference for PCRE that could be used in the present document. Instead, we opt for XSD regular expressions for now. There is precedent for that choice in the IETF, e.g., in YANG [RFC7950].

Note that CDDL uses controls as its main extension point. This creates the opportunity to add further regular expression formats in addition to the one referenced here if desired. As an example, a control ".pcre" is defined in [I-D.bormann-cbor-cddl-freezer].

3.8.4. Control operators .cbor and .cborseq

A ".cbor" control on a byte string indicates that the byte string carries a CBOR encoded data item. Decoded, the data item matches the type given as the right-hand side argument (type1 in the following example).

```
"bytes .cbor type1"
```

Similarly, a ".cborseq" control on a byte string indicates that the byte string carries a sequence of CBOR encoded data items. When the data items are taken as an array, the array matches the type given as the right-hand side argument (type2 in the following example).

```
"bytes .cborseq type2"
```

(The conversion of the encoded sequence to an array can be effected for instance by wrapping the byte string between the two bytes 0x9f and 0xff and decoding the wrapped byte string as a CBOR encoded data item.)

3.8.5. Control operators .within and .and

A ".and" control on a type indicates that the data item matches both that left hand side type and the type given as the right hand side. (Formally, the resulting type is the intersection of the two types given.)

```
"type1 .and type2"
```

A variant of the ".and" control is the ".within" control, which expresses an additional intent: the left hand side type is meant to be a subset of the right-hand-side type.

"type1 .within type2"

While both forms have the identical formal semantics (intersection), the intention of the ".within" form is that the right hand side gives guidance to the types allowed on the left hand side, which typically is a socket (Section 3.9):

```
message = $message .within message-structure
message-structure = [message_type, *message_option]
message_type = 0..255
message_option = any

$message /= [3, dough: text, topping: [* text]]
$message /= [4, noodles: text, sauce: text, parmesan: bool]
```

For ".within", a tool might flag an error if type1 allows data items that are not allowed by type2. In contrast, for ".and", there is no expectation that type1 already is a subset of type2.

3.8.6. Control operators .lt, .le, .gt, .ge, .eq, .ne, and .default

The controls .lt, .le, .gt, .ge, .eq, .ne specify a constraint on the left hand side type to be a value less than, less than or equal, equal to, not equal to, greather than, or greater than or equal to a value given as a (single-valued) right hand side type. In the present specification, the first four controls (.lt, .le, .gt, .ge) are defined only for numeric types, as these have a natural ordering relationship.

```
speed = number .ge 0  ; unit: m/s
```

A variant of the ".ne" control is the ".default" control, which expresses an additional intent: the value specified by the right-hand-side type is intended as a default value for the left hand side type given, and the implied .ne control is there to prevent this value from being sent over the wire. This control is only meaningful when the controld type is used in an optional context; otherwise there would be no way to express the default value.

```
timer = {
  time: uint,
  ? displayed-step: (number .gt 0) .default 1
}
```

3.9. Socket/Plug

Both for type choices and group choices, a mechanism is defined that facilitates starting out with empty choices and assembling them later, potentially in separate files that are concatenated to build the full specification.

Per convention, CDDL extension points are marked with a leading dollar sign (types) or two leading dollar signs (groups). Tools honor that convention by not raising an error if such a type or group is not defined at all; the symbol is then taken to be an empty type choice (group choice), i.e., no choice is available.

```
tcp-header = {seq: uint, ack: uint, * $$tcp-option}

; later, in a different file

$$tcp-option //= (
sack: [+(left: uint, right: uint)]
)

; and, maybe in another file

$$tcp-option //= (
sack-permitted: true
)
```

Names that start with a single "$" are "type sockets", names with a double "$$" are "group sockets". It is not an error if there is no definition for a socket at all; this then means there is no way to satisfy the rule (i.e., the choice is empty).

All definitions (plugs) for socket names must be augmentations, i.e., they must be using "/=" and "//=", respectively.

To pick up the example illustrated in Figure 6, the socket/plug mechanism could be used as shown in Figure 11:

```
PersonalData = {
  ? displayName: tstr,
  NameComponents,
  ? age: uint,
  * $$personaldata-extensions
}

NameComponents = (
  ? firstName: tstr,
  ? familyName: tstr,
)

; The above already works as is.
; But then, we can add later:

$$personaldata-extensions //= (
  favorite-salsa: tstr,
)

; and again, somewhere else:

$$personaldata-extensions //= (
  shoesize: uint,
)
```

Figure 11: Personal Data example: Using socket/plug extensibility

3.10. Generics

Using angle brackets, the left hand side of a rule can add formal parameters after the name being defined, as in:

```
messages = message<"reboot", "now"> / message<"sleep", 1..100>
message<t, v> = {type: t, value: v}
```

When using a generic rule, the formal parameters are bound to the actual arguments supplied (also using angle brackets), within the scope of the generic rule (as if there were a rule of the form parameter = argument).

(There are some limitations to nesting of generics in Appendix F at this time.)

3.11. Operator Precedence

As with any language that has multiple syntactic features such as prefix and infix operators, CDDL has operators that bind more tightly than others. This is becoming more complicated than, say, in ABNF,

as CDDL has both types and groups, with operators that are specific to these concepts. Type operators (such as "/" for type choice) operate on types, while group operators (such as "//" for group choice) operate on groups. Types can simply be used in groups, but groups need to be bracketed (as arrays or maps) to become types. So, type operators naturally bind closer than group operators.

For instance, in

```
t = [group1]
group1 = (a / b // c / d)
a = 1 b = 2 c = 3 d = 4
```

group1 is a group choice between the type choice of a and b and the type choice of c and d. This becomes more relevant once member keys and/or occurrences are added in:

```
t = {group2}
group2 = (? ab: a / b // cd: c / d)
a = 1 b = 2 c = 3 d = 4
```

is a group choice between the optional member "ab" of type a or b and the member "cd" of type c or d. Note that the optionality is attached to the first choice ("ab"), not to the second choice.

Similarly, in

```
t = [group3]
group3 = (+ a / b / c)
a = 1 b = 2 c = 3
```

group3 is a repetition of a type choice between a, b, and c [unflex]; if just a is to be repeatable, a group choice is needed to focus the occurrence:

```
t = [group4]
group4 = (+ a // b / c)
a = 1 b = 2 c = 3
```

group4 is a group choice between a repeatable a and a single b or c.

In general, as with many other languages with operator precedence rules, it is best not to rely on them, but to insert parentheses for readability:

```
t = [group4a]
group4a = ((+ a) // (b / c))
a = 1 b = 2 c = 3
```

The operator precedences, in sequence of loose to tight binding, are defined in Appendix B and summarized in Table 1. (Arities given are 1 for unary prefix operators and 2 for binary infix operators.)

Operator	Ar	Operates on	Prec
=	2	name = type, name = group	1
/=	2	name /= type	1
//=	2	name //= group	1
//	2	group // group	2
,	2	group, group	3
*	1	* group	4
N*M	1	N*M group	4
+	1	+ group	4
?	1	? group	4
=>	2	type => type	5
:	2	name: type	5
/	2	type / type	6
&	1	&group	6
..	2	type..type	7
...	2	type...type	7
.anno	2	type .anno type	7

Table 1: Summary of operator precedences

4. Making Use of CDDL

In this section, we discuss several potential ways to employ CDDL.

4.1. As a guide to a human user

CDDL can be used to efficiently define the layout of CBOR data, such that a human implementer can easily see how data is supposed to be encoded.

Since CDDL maps parts of the CBOR data to human readable names, tools could be built that use CDDL to provide a human friendly representation of the CBOR data, and allow them to edit such data while remaining compliant to its CDDL definition.

4.2. For automated checking of CBOR data structure

CDDL has been specified such that a machine can handle the CDDL definition and related CBOR data (and, thus, also JSON data). For example, a machine could use CDDL to check whether or not CBOR data is compliant to its definition.

The need for thoroughness of such compliance checking depends on the application. For example, an application may decide not to check the data structure at all, and use the CDDL definition solely as a means to indicate the structure of the data to the programmer.

On the other end, the application may also implement a checking mechanism that goes as far as checking that all mandatory map members are available.

The matter in how far the data description must be enforced by an application is left to the designers and implementers of that application, keeping in mind related security considerations.

In no case the intention is that a CDDL tool would be "writing code" for an implementation.

4.3. For data analysis tools

In the long run, it can be expected that more and more data will be stored using the CBOR data format.

Where there is data, there is data analysis and the need to process such data automatically. CDDL can be used for such automated data processing, allowing tools to verify data, clean it, and extract particular parts of interest from it.

Since CBOR is designed with constrained devices in mind, a likely use of it would be small sensors. An interesting use would thus be automated analysis of sensor data.

5. Security considerations

This document presents a content rules language for expressing CBOR data structures. As such, it does not bring any security issues on itself, although specification of protocols that use CBOR naturally need security analysis when defined.

Topics that could be considered in a security considerations section that uses CDDL to define CBOR structures include the following:

- Where could the language maybe cause confusion in a way that will enable security issues?

6. IANA considerations

This document does not require any IANA registrations.

7. References

7.1. Normative References

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[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>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[W3C.REC-xmlschema-2-20041028] Biron, P. and A. Malhotra, "XML Schema Part 2: Datatypes Second Edition", World Wide Web Consortium Recommendation REC-xmlschema-2-20041028, October 2004, <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028>.

7.2. Informative References

[I-D.bormann-cbor-cddl-freezer] Bormann, C., "A feature freezer for the Concise Data Definition Language (CDDL)", draft-bormann-cbor-cddl-freezer-00 (work in progress), January 2018.

[I-D.ietf-anima-grasp] Bormann, C., Carpenter, B., and B. Liu, "A Generic Autonomic Signaling Protocol (GRASP)", draft-ietf-anima-grasp-15 (work in progress), July 2017.

[I-D.ietf-core-senml]
Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Media Types for Sensor Measurement Lists (SenML)", draft-ietf-core-senml-12 (work in progress), December 2017.

[I-D.newton-json-content-rules]
Newton, A. and P. Cordell, "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-09 (work in progress), September 2017.

[RELAXNG] ISO/IEC, "Information technology -- Document Schema Definition Language (DSDL) -- Part 2: Regular-grammar-based validation -- RELAX NG", ISO/IEC 19757-2, December 2008.

[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>.

[RFC7071] Borenstein, N. and M. Kucherawy, "A Media Type for Reputation Interchange", RFC 7071, DOI 10.17487/RFC7071, November 2013, <https://www.rfc-editor.org/info/rfc7071>.

[RFC7950] Bjorklund, M., Ed., "The YANG 1.1 Data Modeling Language", RFC 7950, DOI 10.17487/RFC7950, August 2016, <https://www.rfc-editor.org/info/rfc7950>.

[RFC8007] Murray, R. and B. Niven-Jenkins, "Content Delivery Network Interconnection (CDNI) Control Interface / Triggers", RFC 8007, DOI 10.17487/RFC8007, December 2016, <https://www.rfc-editor.org/info/rfc8007>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

7.3. URIs

[1] https://github.com/cabo/cbor-diag

Appendix A. (Not used.)

Appendix B. ABNF grammar

The following is a formal definition of the CDDL syntax in Augmented Backus-Naur Form (ABNF, [RFC5234]). [_abnftodo]

```
   cddl = S 1*rule
   rule = typename [genericparm] S assign S type S
        / groupname [genericparm] S assign S grpent S

   typename = id
   groupname = id

   assign = "=" / "/=" / "//="

   genericparm = "<" S id S *("," S id S ) ">"
   genericarg = "<" S type1 S *("," S type1 S ) ">"

   type = type1 S *("/" S type1 S)

   type1 = type2 [S (rangeop / ctlop) S type2]

   type2 = value
         / typename [genericarg]
         / "(" type ")"
         / "~" S groupname [genericarg]
         / "#" "6" ["." uint] "(" S type S ")" ; note no space!
         / "#" DIGIT ["." uint]                ; major/ai
         / "#"                                 ; any
         / "{" S group S "}"
         / "[" S group S "]"
         / "&" S "(" S group S ")"
         / "&" S groupname [genericarg]

   rangeop = "..." / ".."

   ctlop = "." id

   group = grpchoice S *("//" S grpchoice S)

   grpchoice = *grpent

   grpent = [occur S] [memberkey S] type optcom
          / [occur S] groupname [genericarg] optcom ; preempted by above
          / [occur S] "(" S group S ")" optcom

   memberkey = type1 S ["^" S] "=>"
             / bareword S ":"
             / value S ":"

   bareword = id

   optcom = S ["," S]
```

```
occur = [uint] "*" [uint]
      / "+"
      / "?"

uint = ["0x" / "0b"] "0"
     / DIGIT1 *DIGIT
     / "0x" 1*HEXDIG
     / "0b" 1*BINDIG

value = number
      / text
      / bytes

int = ["-"] uint

; This is a float if it has fraction or exponent; int otherwise
number = hexfloat / (int ["." fraction] ["e" exponent ])
hexfloat = "0x" 1*HEXDIG ["." 1*HEXDIG] "p" exponent
fraction = 1*DIGIT
exponent = ["+"/"-"] 1*DIGIT

text = %x22 *SCHAR %x22
SCHAR = %x20-21 / %x23-5B / %x5D-10FFFD / SESC
SESC = "\" %x20-10FFFD

bytes = [bsqual] %x27 *BCHAR %x27
BCHAR = %x20-26 / %x28-5B / %x5D-10FFFD / SESC / CRLF
bsqual = %x68 ; "h"
       / %x62.36.34 ; "b64"

id = EALPHA *(*("-" / ".") (EALPHA / DIGIT))
ALPHA = %x41-5A / %x61-7A
EALPHA = %x41-5A / %x61-7A / "@" / "_" / "$"
DIGIT = %x30-39
DIGIT1 = %x31-39
HEXDIG = DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
BINDIG = %x30-31

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

Figure 12: CDDL ABNF

Appendix C. Matching rules

In this appendix, we go through the ABNF syntax rules defined in Appendix B and briefly describe the matching semantics of each syntactic feature. In this context, an instance (data item) "matches" a CDDL specification if it is allowed by the CDDL specification; this is then broken down to parts of specifications (type and group expressions) and parts of instances (data items).

```
cddl = S 1*rule
```

A CDDL specification is a sequence of one or more rules. Each rule gives a name to a right hand side expression, either a CDDL type or a CDDL group. Rule names can be used in the rule itself and/or other rules (and tools can output warnings if that is not the case). The order of the rules is significant only in two cases, including the following: The first rule defines the semantics of the entire specification; hence, its name may be descriptive only (or may be used in itself or other rules as with the other rule names).

```
rule = typename [genericparm] S assign S type S
     / groupname [genericparm] S assign S grpent S

typename = id
groupname = id
```

A rule defines a name for a type expression (production "type") or for a group expression (production "grpent"), with the intention that the semantics does not change when the name is replaced by its (parenthesized if needed) definition.

```
assign = "=" / "/=" / "//="
```

A plain equals sign defines the rule name as the equivalent of the expression to the right. A "/=" or "//=" extends a named type or a group by additional choices; a number of these could be replaced by collecting all the right hand sides and creating a single rule with a type choice or a group choice built from the right hand sides in the order of the rules given. (It is not an error to extend a rule name that has not yet been defined; this makes the right hand side the first entry in the choice being created.) The creation of the type choices and group choices from the right hand sides of rules is the other case where rule order can be significant.

```
genericparm = "<" S id S *("," S id S ) ">"
genericarg = "<" S type1 S *("," S type1 S ) ">"
```

Rule names can have generic parameters, which cause temporary assignments within the right hand sides to the parameter names from the arguments given when citing the rule name.

type = type1 S *("/" S type1 S)

A type can be given as a choice between one or more types. The choice matches a data item if the data item matches any one of the types given in the choice. The choice uses Parse Expression Grammar (PEG) semantics: The first choice that matches wins. (As a result, the order of rules that contribute to a single rule name can very well matter.)

type1 = type2 [S (rangeop / ctlop) S type2]

Two types can be combined with a range operator (which see below) or a control operator (see Section 3.8).

type2 = value

A type can be just a single value (such as 1 or "icecream" or h'0815'), which matches only a data item with that specific value (no conversions defined),

/ typename [genericarg]

or be defined by a rule giving a meaning to a name (possibly after supplying generic args as required by the generic parameters),

/ "(" type ")"

or be defined in a parenthesized type expression (parentheses may be necessary to override some operator precendence), or

/ "~" S groupname [genericarg]

an "unwrapped" group (see Section 3.7), which matches the group inside a type defined as a map or an array by wrapping the group, or

/ "#" "6" ["." uint] "(" S type S ")" ; note no space!

a tagged data item, tagged with the "uint" given and containing the type given as the tagged value, or

/ "#" DIGIT ["." uint] ; major/ai

a data item of a major type (given by the DIGIT), optionally constrained to the additional information given by the uint, or

```
      / "#"                                ; any
```

any data item, or

```
      / "{" S group S "}"
```

a map expression, which matches a valid CBOR map the key/value pairs of which can be ordered in such a way that the resulting sequence matches the group expression, or

```
      / "[" S group S "]"
```

an array expression, which matches a CBOR array the elements of which, when taken as values and complemented by a wildcard (matches anything) key each, match the group, or

```
      / "&" S "(" S group S ")"
      / "&" S groupname [genericarg]
```

an enumeration expression, which matches any a value that is within the set of values that the values of the group given can take.

```
rangeop = "..." / ".."
```

A range operator can be used to join two type expressions that stand for either two integer values or two floating point values; it matches any value that is between the two values, where the first value is always included in the matching set and the second value is included for ".." and excluded for "...".

```
ctlop = "." id
```

A control operator ties a _target_ type to a _controller_ type as defined in Section 3.8. Note that control operators are an extension point for CDDL; additional documents may want to define additional control operators.

```
group = grpchoice S *("//" S grpchoice S)
```

A group matches any sequence of key/value pairs that matches any of the choices given (again using Parse Expression Grammar semantics).

```
grpchoice = *grpent
```

Each of the component groups is given as a sequence of group entries. For a match, the sequence of key/value pairs given needs to match the sequence of group entries in the sequence given.

```
grpent = [occur S] [memberkey S] type optcom
```

A group entry can be given by a value type, which needs to be matched by the value part of a single element, and optionally a memberkey type, which needs to be matched by the key part of the element, if the memberkey is given. If the memberkey is not given, the entry can only be used for matching arrays, not for maps. (See below how that is modified by the occurrence indicator.)

```
       / [occur S] groupname [genericarg] optcom ; preempted by above
```

A group entry can be built from a named group, or

```
       / [occur S] "(" S group S ")" optcom
```

from a parenthesized group, again with a possible occurrence indicator.

```
memberkey = type1 S ["^" S] "=>"
          / bareword S ":"
          / value S ":"
```

Key types can be given by a type expression, a bareword (which stands for string value created from this bareword), or a value (which stands for a type that just contains this value). A key value matches its key type if the key value is a member of the key type, unless a cut preceding it in the group applies (see Section 3.5.3 how map matching is infuenced by the presence of the cuts denoted by "^" or ":" in previous entries).

```
bareword = id
```

A bareword is an alternative way to write a type with a single text string value; it can only be used in the syntactic context given above.

```
optcom = S ["," S]
```

(Optional commas do not influence the matching.)

```
occur = [uint] "*" [uint]
      / "+"
      / "?"
```

An occurrence indicator modifies the group given to its right by requiring the group to match the sequence to be matched exactly for a certain number of times (see Section 3.2) in sequence, i.e. it acts

as a (possibly infinite) group choice that contains choices with the group repeated each of the occurrences times.

The rest of the ABNF describes syntax for value notation that should be familiar from programming languages, with the possible exception of h'..' and b64'..' for byte strings, as well as syntactic elements such as comments and line ends.

Appendix D. (Not used.)

Appendix E. Standard Prelude

The following prelude is automatically added to each CDDL file [tdate]. (Note that technically, it is a postlude, as it does not disturb the selection of the first rule as the root of the definition.)

```
any = #

uint = #0
nint = #1
int = uint / nint

bstr = #2
bytes = bstr
tstr = #3
text = tstr

tdate = #6.0(tstr)
time = #6.1(number)
number = int / float
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
decfrac = #6.4([e10: int, m: integer])
bigfloat = #6.5([e2: int, m: integer])
eb64url = #6.21(any)
eb64legacy = #6.22(any)
eb16 = #6.23(any)
encoded-cbor = #6.24(bstr)
uri = #6.32(tstr)
b64url = #6.33(tstr)
b64legacy = #6.34(tstr)
regexp = #6.35(tstr)
mime-message = #6.36(tstr)
cbor-any = #6.55799(any)

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
undefined = #7.23
```

Figure 13: CDDL Prelude

Note that the prelude is deemed to be fixed. This means, for instance, that additional tags beyond [RFC7049], as registered, need to be defined in each CDDL file that is using them.

A common stumbling point is that the prelude does not define a type "string". CBOR has byte strings ("bytes" in the prelude) and text strings ("text"), so a type that is simply called "string" would be ambiguous.

E.1. Use with JSON

The JSON generic data model (implicit in [RFC8259]) is a subset of the generic data model of CBOR. So one can use CDDL with JSON by limiting oneself to what can be represented in JSON. Roughly speaking, this means leaving out byte strings, tags, and simple values other than "false", "true", and "null", leading to the following limited prelude:

```
any = #

uint = #0
nint = #1
int = uint / nint

tstr = #3
text = tstr

number = int / float

float16 = #7.25
float32 = #7.26
float64 = #7.27
float16-32 = float16 / float32
float32-64 = float32 / float64
float = float16-32 / float64

false = #7.20
true = #7.21
bool = false / true
nil = #7.22
null = nil
```

Figure 14: JSON compatible subset of CDDL Prelude

(The major types given here do not have a direct meaning in JSON, but they can be interpreted as CBOR major types translated through Section 4 of [RFC7049].)

There are a few fine points in using CDDL with JSON. First, JSON does not distinguish between integers and floating point numbers; there is only one kind of number (which may happen to be integral). In this context, specifying a type as "uint", "nint" or "int" then becomes a predicate that the number be integral. As an example, this means that the following JSON numbers are all matching "uint":

10 10.0 1e1 1.0e1 100e-1

(The fact that these are all integers may be surprising to users accustomed to the long tradition in programming languages of using decimal points or exponents in a number to indicate a floating point literal.)

CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. The effect of specifying a floating point precision (float16/float32/float64) is only to restrict the set of permissible values to those expressible with binary16/binary32/binary64; this is unlikely to be very useful when using CDDL for specifying JSON data structures.

Fundamentally, the number system of JSON itself is based on decimal numbers and decimal fractions and does not have limits to its precision or range. In practice, JSON numbers are often parsed into a number type that is called float64 here, creating a number of limitations to the generic data model [RFC7493]. In particular, this means that integers can only be expressed with interoperable exactness when they lie in the range [-(2**53)+1, (2**53)-1] -- a smaller range than that covered by CDDL "int".

JSON applications that want to stay compatible with I-JSON therefore may want to define integer types with more limited ranges, such as in Figure 15. Note that the types given here are not part of the prelude; they need to be copied into the CDDL specification if needed.

```
ij-uint = 0..9007199254740991
ij-nint = -9007199254740991..-1
ij-int = -9007199254740991..9007199254740991
```

Figure 15: I-JSON types for CDDL (not part of prelude)

JSON applications that do not need to stay compatible with I-JSON and that actually may need to go beyond the 64-bit unsigned and negative integers supported by "int" (= "uint"/"nint") may want to use the following additional types from the standard prelude, which are expressed in terms of tags but can straightforwardly be mapped into JSON (but not I-JSON) numbers:

```
biguint = #6.2(bstr)
bignint = #6.3(bstr)
bigint = biguint / bignint
integer = int / bigint
unsigned = uint / biguint
```

CDDL at this point does not have a way to express the unlimited floating point precision that is theoretically possible with JSON; at the time of writing, this is rarely used in protocols in practice.

Note that a data model described in CDDL is always restricted by what can be expressed in the serialization; e.g., floating point values such as NaN (not a number) and the infinities cannot be represented in JSON even if they are allowed in the CDDL generic data model.

Appendix F. The CDDL tool

A rough CDDL tool is available. For CDDL specifications, it can check the syntax, generate one or more instances (expressed in CBOR diagnostic notation or in pretty-printed JSON), and validate an existing instance against the specification:

```
Usage:
cddl spec.cddl generate [n]
cddl spec.cddl json-generate [n]
cddl spec.cddl validate instance.cbor
cddl spec.cddl validate instance.json
```

Figure 16: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 17: CDDL tool installation

The accompanying CBOR diagnostic tools (which are automatically installed by the above) are described in https://github.com/cabo/cbor-diag [1]; they can be used to convert between binary CBOR, a pretty-printed form of that, CBOR diagnostic notation, JSON, and YAML.

Appendix G. Extended Diagnostic Notation

Section 6 of [RFC7049] defines a "diagnostic notation" in order to be able to converse about CBOR data items without having to resort to binary data. Diagnostic notation is based on JSON, with extensions for representing CBOR constructs such as binary data and tags.

(Standardizing this together with the actual interchange format does not serve to create another interchange format, but enables the use of a shared diagnostic notation in tools for and documents about CBOR.)

This section discusses a few extensions to the diagnostic notation that have turned out to be useful since RFC 7049 was written. We refer to the result as extended diagnostic notation (EDN).

G.1. White space in byte string notation

Examples often benefit from some white space (spaces, line breaks) in byte strings. In extended diagnostic notation, white space is ignored in prefixed byte strings; for instance, the following are equivalent:

```
h'48656c6c6f20776f726c64'
h'48 65 6c 6c 6f 20 77 6f 72 6c 64'
h'4 86 56c 6c6f
  20776 f726c64'
```

G.2. Text in byte string notation

Diagnostic notation notates Byte strings in one of the [RFC4648] base encodings,, enclosed in single quotes, prefixed by >h< for base16, >b32< for base32, >h32< for base32hex, >b64< for base64 or base64url. Quite often, byte strings carry bytes that are meaningfully interpreted as UTF-8 text. Extended Diagnostic Notation allows the use of single quotes without a prefix to express byte strings with UTF-8 text; for instance, the following are equivalent:

```
'hello world'
h'68656c6c6f20776f726c64'
```

The escaping rules of JSON strings are applied equivalently for text-based byte strings, e.g., \ stands for a single backslash and ' stands for a single quote. White space is included literally, i.e., the previous section does not apply to text-based byte strings.

G.3. Embedded CBOR and CBOR sequences in byte strings

Where a byte string is to carry an embedded CBOR-encoded item, or more generally a sequence of zero or more such items, the diagnostic notation for these zero or more CBOR data items, separated by commata, can be enclosed in << and >> to notate the byte string resulting from encoding the data items and concatenating the result. For instance, each pair of columns in the following are equivalent:

```
<<1>>              h'01'
<<1, 2>>           h'0102'
<<"foo", null>>    h'63666F6FF6'
<<>>               h''
```

G.4. Concatenated Strings

While the ability to include white space enables line-breaking of encoded byte strings, a mechanism is needed to be able to include text strings as well as byte strings in direct UTF-8 representation into line-based documents (such as RFCs and source code).

We extend the diagnostic notation by allowing multiple text strings or multiple byte strings to be notated separated by white space, these are then concatenated into a single text or byte string, respectively. Text strings and byte strings do not mix within such a concatenation, except that byte string notation can be used inside a sequence of concatenated text string notation to encode characters that may be better represented in an encoded way. The following four values are equivalent:

```
"Hello world"
"Hello " "world"
"Hello" h'20' "world"
"" h'48656c6c6f20776f726c64' ""
```

Similarly, the following byte string values are equivalent

```
'Hello world'
'Hello ' 'world'
'Hello ' h'776f726c64'
'Hello' h'20' 'world'
'' h'48656c6c6f20776f726c64' '' b64''
h'4 86 56c 6c6f' h' 20776 f726c64'
```

(Note that the approach of separating by whitespace, while familiar from the C language, requires some attention - a single comma makes a big difference here.)

G.5. Hexadecimal, octal, and binary numbers

In addition to JSON's decimal numbers, EDN provides hexadecimal, octal and binary numbers in the usual C-language notation (octal with 0o prefix present only).

The following are equivalent:

```
4711
0x1267
0o11147
0b1001001100111
```

As are:

```
1.5
0x1.8p0
0x18p-4
```

G.6. Comments

Longer pieces of diagnostic notation may benefit from comments. JSON famously does not provide for comments, and basic RFC 7049 diagnostic notation inherits this property.

In extended diagnostic notation, comments can be included, delimited by slashes ("/"). Any text within and including a pair of slashes is considered a comment.

Comments are considered white space. Hence, they are allowed in prefixed byte strings; for instance, the following are equivalent:

```
h'68656c6c6f20776f726c64'
h'68 65 6c /doubled l!/ 6c 6f /hello/
  20 /space/
  77 6f 72 6c 64' /world/
```

This can be used to annotate a CBOR structure as in:

```
/grasp-message/ [/M_DISCOVERY/ 1, /session-id/ 10584416,
                 /objective/ [/objective-name/ "opsonize",
                              /D, N, S/ 7, /loop-count/ 105]]
```

(There are currently no end-of-line comments. If we want to add them, "//" sounds like a reasonable delimiter given that we already use slashes for comments, but we also could go e.g. for "#".)

Appendix H. Examples

This section contains various examples of structures defined using CDDL.

The theme for the first example is taken from [RFC7071], which defines certain JSON structures in English. For a similar example, it may also be of interest to examine Appendix A of [RFC8007], which

contains a CDDL definition for a JSON structure defined in the main body of the RFC.

The second subsection in this appendix translates examples from [I-D.newton-json-content-rules] into CDDL.

These examples all happen to describe data that is interchanged in JSON. Examples for CDDL definitions of data that is interchanged in CBOR can be found in [RFC8152], [I-D.ietf-anima-grasp], or [I-D.ietf-core-senml].

H.1. RFC 7071

[RFC7071] defines the Reputon structure for JSON using somewhat formalized English text. Here is a (somewhat verbose) equivalent definition using the same terms, but notated in CDDL:

```
reputation-object = {
  reputation-context,
  reputon-list
}

reputation-context = (
  application: text
)

reputon-list = (
  reputons: reputon-array
)

reputon-array = [* reputon]

reputon = {
  rater-value,
  assertion-value,
  rated-value,
  rating-value,
  ? conf-value,
  ? normal-value,
  ? sample-value,
  ? gen-value,
  ? expire-value,
  * ext-value,
}

rater-value = ( rater: text )
assertion-value = ( assertion: text )
rated-value = ( rated: text )
rating-value = ( rating: float16 )
conf-value = ( confidence: float16 )
normal-value = ( normal-rating: float16 )
sample-value = ( sample-size: uint )
gen-value = ( generated: uint )
expire-value = ( expires: uint )
ext-value = ( text => any )
```

An equivalent, more compact form of this example would be:

```
reputation-object = {
  application: text
  reputons: [* reputon]
}

reputon = {
  rater: text
  assertion: text
  rated: text
  rating: float16
  ? confidence: float16
  ? normal-rating: float16
  ? sample-size: uint
  ? generated: uint
  ? expires: uint
  * text => any
}
```

Note how this rather clearly delineates the structure somewhat shrouded by so many words in section 6.2.2. of [RFC7071]. Also, this definition makes it clear that several ext-values are allowed (by definition with different member names); RFC 7071 could be read to forbid the repetition of ext-value ("A specific reputon-element MUST NOT appear more than once" is ambiguous.)

The CDDL tool (which hasn't quite been trained for polite conversation) says:

```
{
  "application": "tridentiferous",
  "reputons": [
    {
      "rater": "loamily",
      "assertion": "Dasyprocta",
      "rated": "uncommensurableness",
      "rating": 0.05055809746548934,
      "confidence": 0.7484706448605812,
      "normal-rating": 0.8677887734049299,
      "sample-size": 4059,
      "expires": 3969,
      "bearer": "nitty",
      "faucal": "postulnar",
      "naturalism": "sarcotic"
    },
    {
      "rater": "precreed",
      "assertion": "xanthosis",
      "rated": "balsamy",
      "rating": 0.36091333590593955,
      "confidence": 0.3700759808403371,
      "sample-size": 3904
    },
    {
      "rater": "urinosexual",
      "assertion": "malacostracous",
      "rated": "arenariae",
      "rating": 0.9210673488013762,
      "normal-rating": 0.4778762617112776,
      "sample-size": 4428,
      "generated": 3294,
      "backfurrow": "enterable",
      "fruitgrower": "flannelflower"
    },
    {
      "rater": "pedologistically",
      "assertion": "unmetaphysical",
      "rated": "elocutionist",
      "rating": 0.42073613384304287,
      "misimagine": "retinaculum",
      "snobbish": "contradict",
      "Bosporanic": "periostotomy",
      "dayworker": "intragyral"
    }
  ]
}
```

H.1.1. Examples from JSON Content Rules

Although JSON Content Rules [I-D.newton-json-content-rules] seems to address a more general problem than CDDL, it is still a worthwhile resource to explore for examples (beyond all the inspiration the format itself has had for CDDL).

Figure 2 of the JCR I-D looks very similar, if slightly less noisy, in CDDL:

```
root = [2*2 {
  precision: text,
  Latitude: float,
  Longitude: float,
  Address: text,
  City: text,
  State: text,
  Zip: text,
  Country: text
}]
```

Figure 18: JCR, Figure 2, in CDDL

Apart from the lack of a need to quote the member names, text strings are called "text" or "tstr" in CDDL ("string" would be ambiguous as CBOR also provides byte strings).

The CDDL tool creates the below example instance for this:

```
 [{"precision": "pyrosphere", "Latitude": 0.5399712314350172,
   "Longitude": 0.5157523963028087, "Address": "resow",
   "City": "problemwise", "State": "martyrlike", "Zip": "preprove",
   "Country": "Pace"},
  {"precision": "unrigging", "Latitude": 0.10422704368372193,
   "Longitude": 0.6279808663725834, "Address": "picturedom",
   "City": "decipherability", "State": "autometry", "Zip": "pout",
   "Country": "wimple"}]
```

Figure 4 of the JCR I-D in CDDL:

```
root = { image }

image = (
  Image: {
    size,
    Title: text,
    thumbnail,
    IDs: [* int]
  }
)

size = (
  Width: 0..1280
  Height: 0..1024
)

thumbnail = (
  Thumbnail: {
    size,
    Url: ~uri
  }
)
```

This shows how the group concept can be used to keep related elements (here: width, height) together, and to emulate the JCR style of specification. (It also shows referencing a type by unwrapping a tag from the prelude, "uri" - this could be done differently.) The more compact form of Figure 5 of the JCR I-D could be emulated like this:

```
root = {
  Image: {
    size, Title: text,
    Thumbnail: { size, Url: ~uri },
    IDs: [* int]
  }
}

size = (
  Width: 0..1280,
  Height: 0..1024,
)
```

The CDDL tool creates the below example instance for this:

```
{"Image": {"Width": 566, "Height": 516, "Title": "leisterer",
  "Thumbnail": {"Width": 1111, "Height": 176, "Url": 32("scrog")},
  "IDs": []}}
```

Acknowledgements

CDDL was originally conceived by Bert Greevenbosch, who also wrote the original five versions of this document.

Inspiration was taken from the C and Pascal languages, MPEG's conventions for describing structures in the ISO base media file format, Relax-NG and its compact syntax [RELAXNG], and in particular from Andrew Lee Newton's "JSON Content Rules" [I-D.newton-json-content-rules].

Useful feedback came from members of the IETF CBOR WG, in particular Joe Hildebrand, Sean Leonard and Jim Schaad. Also, Francesca Palombini and Joe volunteered to chair this WG, providing the framework for generating and processing this feedback.

The CDDL tool was written by Carsten Bormann, building on previous work by Troy Heninger and Tom Lord.

Editorial Comments

[_format] So far, the ability to restrict format choices have not been needed beyond the floating point formats. Those can be applied to ranges using the new .and control now. It is not clear we want to add more format control before we have a use case.

[_range] TO DO: define this precisely. This clearly includes integers and floats. Strings - as in "a".."z" - could be added if desired, but this would require adopting a definition of string ordering and possibly a successor function so "a".."z" does not include "bb".

[_strings] TO DO: This still needs to be fully realized in the ABNF and in the CDDL tool.

[_bitsendian] How useful would it be to have another variant that counts bits like in RFC box notation? (Or at least per-byte? 32-bit words don't always perfectly mesh with byte strings.)

[unflex] A comment has been that this is counter-intuitive. One solution would be to simply disallow unparenthesized usage of occurrence indicators in front of type choices unless a member key is also present like in group2 above.

[_abnftodo] Potential improvements: the prefixed byte strings are more liberally specified than they actually are.

[tdate] The prelude as included here does not yet have a .regexp control
on tdate, but we probably do want to have one.

Authors' Addresses

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

Carsten Bormann
Universitaet Bremen TZI
Bibliothekstr. 1
Bremen D-28359
Germany

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