Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 20, 2016

C. Vigano
Universitaet Bremen
H. Birkholz
Fraunhofer SIT
October 18, 2015

CBOR data definition language (CDDL): a notational convention to express CBOR data structures

draft-greevenbosch-appsawg-cbor-cddl-07

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 http://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 April 20, 2016.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

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 5
2.1.1. Usage 7
2.1.2. Syntax 8
2.2. Types 8
2.2.1. Values 8
2.2.2. Choices 8
2.2.3. Representation Types 10
2.2.4. Root type 10
3. Syntax 11
3.1. General conventions 11
3.2. Occurrence 12
3.3. Predefined names for types 12
3.4. Arrays 13
3.5. Maps 14
3.5.1. Structs 14
3.5.2. Tables 17
3.6. Tags 17
3.7. Operator Precedence 18
4. Examples 19
4.1. Moves in a computer game 19
4.2. Fruit 24
4.3. RFC 7071 25
4.4. Examples from JSON Content Rules 29
5. Making Use of CDDL 31
5.1. As a guide to a human user 31
5.2. For automated checking of CBOR data structure 31
5.3. For data analysis tools 31
6. Discussion 32
6.1. Work to do 32
7. Resolved Issues 32
8. Security considerations 32
9. IANA considerations 33
10. Acknowledgements 33
11. References 33
11.1. Normative References 33
11.2. Informative References 34
Appendix A. Cemetery 34
Appendix B. Nursery 34
B.1. Annotations 34

B.1.1. Annotation .size 35
B.1.2. Annotation .bits 35
B.1.3. Annotation .regexp 36
B.1.4. Annotations .cbor and .cborseq 37
B.1.5. Annotations .within and .and 37
B.2. Socket/Plug 38
B.3. Generics 39
Appendix C. Change Log 40
Appendix D. ABNF grammar 41
Appendix E. Standard Prelude 43
Appendix F. The CDDL tool 45
Appendix G. Extended Diagnostic Notation 45
G.1. White space in binary strings 46
G.2. Text in binary strings 46
G.3. Concatenated Strings 46
G.4. Hexadecimal, octal, and binary numbers 47
G.5. Comments 47
Authors' Addresses 49

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 "CBOR 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.

This document has the following structure:

The syntax of CDDL is defined in Section 3. Examples of CDDL and related CBOR data instances are defined in Section 4. Section 5 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 D. 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), 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 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 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 instances that are acceptable 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.

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 group can be placed in (round) parentheses, and given a name by using it in a rule:

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

Figure 1: A basic group

Or a group can just be used in the definition of something else:

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

Figure 2: Using a group in a map

which, given the above rule for pii, is identical to:

```
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.

The parentheses for groups are optional when there is some other set of brackets present, so it would be slightly more natural to express Figure 2 as:

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

Groups can be used to factor out common parts of structs, e.g., instead of writing:

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

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

one can choose a name for the common subgroup and write:

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

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

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

Figure 4: Using a group for factorization

Note that the contents of 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 string 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 valuetype. This 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 either a valuetype derived from the major types defined in [RFC7049], a convenience valuetype defined in this document (Appendix E) or the name of a group defined in the protocol file.

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 D 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 a mistake 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.

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 D 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'.

- Strings are enclosed by double quotation '"' characters. They follow the conventions for strings as defined in [RFC7159], section 7. [_strings]

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

Example:

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

g = (
  "name": tstr,
  age: int,
)
```

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.

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" IEEE 754 half-precision float (major type 7, additional information 25).

"float32" IEEE 754 single-precision float (major type 7, additional information 26).

"float64" 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.

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, 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.

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],
}
```

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 5: 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 Appendix B.2 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 = {* x => tstr}
x = int / float
```

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. 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 D 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. Examples

This section contains various examples of structures defined using CDDL.

4.1. Moves in a computer game

A multiplayer computer game uses CBOR to exchange moves between the players. To ensure a good gaming experience, the move information needs to be exchanged quickly and frequently. Therefore, the game uses CBOR to send its information in a compact format. Figure 6 shows definition of the CBOR information exchange format.

```
UpdateMsg = [* {
  move_no        : uint,                ; increases for each move
  player_info    : PlayerInfo,          ; general information
  moves          : Moves,               ; moves in this message
}]

PlayerInfo = {
  alias          : tstr,
  player_id      : uint,
  experience     : uint,                ; beginner: 0; expert: 3
  gold           : uint,
  supplies       : Supplies,
  avg_strength   : float16,
}

Supplies = {
  wood  => uint
  iron  => uint
  grain  => uint
}

wood = 0
iron = 1
grain = 2

Moves = [* Move]

Move = (
  unit_id        : uint,
  unit_strength  : uint,              ; between 0 and 100
  2*2 source_pos : uint,              ; (x,y)
  2*2 target_pos : uint,              ; (x,y)
)
```

Figure 6: CDDL definition of an information exchange format for a computer game

The CDDL tool generates this as a possible instance:

```
  [{"move_no": 3985, "player_info":
    {"alias": "timbrologist", "player_id": 699, "experience": 2699,
     "gold": 328, "supplies": {0: 1768, 1: 3087, 2: 1401},
     "avg_strength": 0.9712613869888417},
    "moves": [[1702, 458, 38, 399, 327, 304],
              [3145, 4454, 1175, 3441, 74, 1542],
              [4099, 4062, 2808, 8, 3174, 3048],
              [367, 3649, 756, 3644, 3725, 2769]]},
   {"move_no": 199, "player_info":
    {"alias": "cipo", "player_id": 4309, "experience": 4094,
     "gold": 4114, "supplies": {0: 873, 1: 4706, 2: 1733},
     "avg_strength": 0.37808379403466696},
    "moves": [[1977, 3129, 3890, 4000, 1555, 377],
              [2646, 286, 3363, 4381, 3815, 1039]]},
   {"move_no": 2226, "player_info":
    {"alias": "Stacey", "player_id": 1055, "experience": 207,
     "gold": 285, "supplies": {0: 3325, 1: 1515, 2: 3304},
     "avg_strength": 0.8590028130444863},
    "moves": [[869, 4126, 2382, 3155, 1523, 2621]]}]
```

Notice that the supplies have been encoded as a map with integer keys. In this example, using string keys would also have been suitable; the example just illustrates the possibility to use other datatypes for keys, leading to more efficient encoding.

The tool-generated binary CBOR for the instance about cannot express yet that the floating point values are 16-bit:

```
83                                  # array(3)
   a3                               # map(3)
      67                            # text(7)
         6d6f76655f6e6f             # "move_no"
      19 0f91                       # unsigned(3985)
      6b                            # text(11)
         706c617965725f696e666f     # "player_info"
      a6                            # map(6)
         65                         # text(5)
            616c696173              # "alias"
         6c                         # text(12)
            74696d62726f6c6f67697374 # "timbrologist"
         69                         # text(9)
            706c617965725f6964      # "player_id"
         19 02bb                    # unsigned(699)
         6a                         # text(10)
            657870657269656e6365    # "experience"
         19 0a8b                    # unsigned(2699)
         64                         # text(4)
            676f6c64                # "gold"
```

```
            19 0148                    # unsigned(328)
            68                         # text(8)
               737570706c696573         # "supplies"
            a3                         # map(3)
               00                      # unsigned(0)
               19 06e8                 # unsigned(1768)
               01                      # unsigned(1)
               19 0c0f                 # unsigned(3087)
               02                      # unsigned(2)
               19 0579                 # unsigned(1401)
            6c                         # text(12)
               6176675f737472656e677468 # "avg_strength"
            fb 3fef1492c29f8275        # primitive(4606923564386321013)
         65                            # text(5)
            6d6f766573                 # "moves"
         84                            # array(4)
            86                         # array(6)
               19 06a6                 # unsigned(1702)
               19 01ca                 # unsigned(458)
               18 26                   # unsigned(38)
               19 018f                 # unsigned(399)
               19 0147                 # unsigned(327)
               19 0130                 # unsigned(304)
            86                         # array(6)
               19 0c49                 # unsigned(3145)
               19 1166                 # unsigned(4454)
               19 0497                 # unsigned(1175)
               19 0d71                 # unsigned(3441)
               18 4a                   # unsigned(74)
               19 0606                 # unsigned(1542)
            86                         # array(6)
               19 1003                 # unsigned(4099)
               19 0fde                 # unsigned(4062)
               19 0af8                 # unsigned(2808)
               08                      # unsigned(8)
               19 0c66                 # unsigned(3174)
               19 0be8                 # unsigned(3048)
            86                         # array(6)
               19 016f                 # unsigned(367)
               19 0e41                 # unsigned(3649)
               19 02f4                 # unsigned(756)
               19 0e3c                 # unsigned(3644)
               19 0e8d                 # unsigned(3725)
               19 0ad1                 # unsigned(2769)
      a3                               # map(3)
         67                            # text(7)
            6d6f76655f6e6f             # "move_no"
         18 c7                         # unsigned(199)
```

```
      6b                          # text(11)
         706c617965725f696e666f   # "player_info"
      a6                          # map(6)
         65                       # text(5)
            616c696173            # "alias"
         64                       # text(4)
            6369706f              # "cipo"
         69                       # text(9)
            706c617965725f6964    # "player_id"
         19 10d5                  # unsigned(4309)
         6a                       # text(10)
            657870657269656e6365  # "experience"
         19 0ffe                  # unsigned(4094)
         64                       # text(4)
            676f6c64              # "gold"
         19 1012                  # unsigned(4114)
         68                       # text(8)
            737570706c696573      # "supplies"
         a3                       # map(3)
            00                    # unsigned(0)
            19 0369               # unsigned(873)
            01                    # unsigned(1)
            19 1262               # unsigned(4706)
            02                    # unsigned(2)
            19 06c5               # unsigned(1733)
         6c                       # text(12)
            6176675f737472656e677468 # "avg_strength"
         fb 3fd832865ea1b216      # primitive(4600482572053623318)
      65                          # text(5)
         6d6f766573               # "moves"
      82                          # array(2)
         86                       # array(6)
            19 07b9               # unsigned(1977)
            19 0c39               # unsigned(3129)
            19 0f32               # unsigned(3890)
            19 0fa0               # unsigned(4000)
            19 0613               # unsigned(1555)
            19 0179               # unsigned(377)
         86                       # array(6)
            19 0a56               # unsigned(2646)
            19 011e               # unsigned(286)
            19 0d23               # unsigned(3363)
            19 111d               # unsigned(4381)
            19 0ee7               # unsigned(3815)
            19 040f               # unsigned(1039)
   a3                             # map(3)
      67                          # text(7)
         6d6f76655f6e6f           # "move_no"
```

```
19 08b2                         # unsigned(2226)
6b                              # text(11)
   706c617965725f696e666f       # "player_info"
a6                              # map(6)
   65                           # text(5)
      616c696173                # "alias"
   66                           # text(6)
      537461636579              # "Stacey"
   69                           # text(9)
      706c617965725f6964        # "player_id"
   19 041f                      # unsigned(1055)
   6a                           # text(10)
      657870657269656e6365      # "experience"
   18 cf                        # unsigned(207)
   64                           # text(4)
      676f6c64                  # "gold"
   19 011d                      # unsigned(285)
   68                           # text(8)
      737570706c696573          # "supplies"
   a3                           # map(3)
      00                        # unsigned(0)
      19 0cfd                   # unsigned(3325)
      01                        # unsigned(1)
      19 05eb                   # unsigned(1515)
      02                        # unsigned(2)
      19 0ce8                   # unsigned(3304)
   6c                           # text(12)
      6176675f737472656e677468  # "avg_strength"
   fb 3feb7cf377a65699          # primitive(4605912429042751129)
65                              # text(5)
   6d6f766573                   # "moves"
81                              # array(1)
   86                           # array(6)
      19 0365                   # unsigned(869)
      19 101e                   # unsigned(4126)
      19 094e                   # unsigned(2382)
      19 0c53                   # unsigned(3155)
      19 05f3                   # unsigned(1523)
      19 0a3d                   # unsigned(2621)
```

Figure 7: CBOR instance for game example

4.2. Fruit

Figure 8 contains an example for a CBOR structure that contains information about fruit.

```
fruitlist = [* Fruit]

Fruit = {
  name                : tstr,
  colour              : [* color],
  avg_weight          : float16,
  price               : uint,
  international_names  : International,
  rfu                 : bstr,          ; reserved for future use
}

International = {
  "DE"                : tstr,          ; German
  "EN"                : tstr,          ; English
  "FR"                : tstr,          ; French
  "NL"                : tstr,          ; Dutch
  "ZH-HANS"           : tstr,          ; Chinese
}

color = &(
  black: 0, red: 1,  green: 2,  yellow: 3,
  blue: 4,  magenta: 5,  cyan: 6,  white: 7,
)
```

Figure 8: Example CBOR structure

4.3. 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"
    }
  ]
}
```

4.4. 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 9: 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 using 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": []}}
```

5. Making Use of CDDL

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

5.1. 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.

5.2. For automated checking of CBOR data structure

CDDL has been specified such that a machine can handle the CDDL definition and related CBOR data. 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 pairs 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.

5.3. 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.

6. Discussion

CDDL already is usable in its present form, as Section 4.3 should have demonstrated. However, additional examples should be developed, and some experience be gained with the usefulness of tools built around CDDL.

6.1. Work to do

- The precise semantics of occurrence indicators as defined in Section 3.2 could be explained in more detail. E.g., the exact semantics of an occurrence indicators on a group name in a map (which means the entire group can occur in this way).

- Build good use cases that, one each, demonstrate vector, record, table and struct usage.

- There probably are some security considerations.

See also the editorial comments sprinkled throughout the document.

7. Resolved Issues

- The key/value pairs in 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.

- CDDL distinguishes the various CBOR number types, but there is only one number type in JSON. There is no effect in specifying a precision (float16/float32/float64) when using CDDL for specifying JSON data structures. (The current validator implementation Appendix F does not handle this very well, either.)

8. 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?

9. IANA considerations

This document does not require any IANA registrations.

10. 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 Carsten Bormann, Joe Hildebrand, Sean Leonard and Jim Schaad.

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

11. References

11.1. 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, <http://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, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc7049>.

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

11.2. Informative References

[RELAXNG] OASIS, "RELAX-NG Compact Syntax", November 2002, <http://relaxng.org/compact-20021121.html>.

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

[I-D.newton-json-content-rules] Newton, A., "A Language for Rules Describing JSON Content", draft-newton-json-content-rules-04 (work in progress), December 2014.

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

Appendix A. Cemetery

The following ideas are buried for now:

- <...> as syntax for enumerations. We view values to be just another type (a very specific type with just one member), so that an enumeration can be denoted as a choice using "/" as the delimiter of choices. Because of this, no evidence is present that a separate syntax for enumerations is needed.

Appendix B. Nursery

This appendix describes advanced features that are still under heavy review.

B.1. Annotations

An _annotation_ allows to annotate a _target_ type with a _control_ type via an _annotator_.

The syntax for an annotated type is "target .annotator control", where annotators are special identifiers prefixed by a dot. (Note that _target_ or _control_ might need to be parenthesized.)

Three annotators are defined at his point. Note that the CDDL tool does not currently support combining multiple annotations on a single target.

B.1.1. Annotation .size

A ".size" annotation 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 10: Annotation for size in bytes

(FIXME: In the CDDL tool, the target must be a byte string for now.)

When applied to an unsigned integer, the ".size" annotation 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 11: Annotation for integer size in bytes

Note that, as with value restrictions in CDDL, this annotation 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]).

B.1.2. Annotation .bits

A ".bits" annotation 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" annotation on an unsigned integer "i" indicates that for all unsigned integers "n" where "(i & (1 << n)) != 0", "n" is 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 12: Annotation 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.

B.1.3. Annotation .regexp

A ".regexp" annotation indicates that the text string given as a target needs to match the PCRE regular expression given as a value in the control type, where that regular expression is anchored on both sides. (If anchoring is not desired for a side, ".*" needs to be inserted there.)

```
nai = tstr .regexp "\\w+@\\w+(\\.\\w+)+"
```

Figure 13: Annotation with a PCRE regexp

The CDDL tool proposes:

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

B.1.4. Annotations .cbor and .cborseq

A ".cbor" annotation 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" annotation 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.)

B.1.5. Annotations .within and .and

A ".and" annotation 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" annotation is the ".within" annotation, 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 (Appendix B.2):

```
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.

B.2. 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 augments, i.e., they must be using "/=" and "//=", respectively.

To pick up the example illustrated in Figure 5, the socket/plug mechanism could be used as shown in Figure 14:

```
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 14: Personal Data example: Using socket/plug extensibility

B.3. 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.)

Appendix C. Change Log

Changes from version 00 to version 01:

- Removed constants
- Updated the tag mechanism
- Extended the map structure
- Added examples

Changes from version 01 to version 02:

- Fixed example

Changes from version 02 to version 03:

- Added information about characters used in names
- Added text about an overall data structure and order of definition of fields
- Added text about encoding of keys
- Added table with keywords
- Strings and integer writing conventions
- Added ABNF

Changes from version 03 to version 04:

- Removed optional fields for non-maps
- Defined all key/value pairs in maps are considered optional from the CDDL perspective
- Allow omission of type of keys for maps with only text string and integer keys
- Changed order of definitions
- Updated fruit and moves examples
- Renamed the "Philosophy" section to "Using CDDL", and added more text about CDDL usage

- Several editorials

Changes from version 04 to version 05:

- Added text about alternative datatypes and any datatype
- Fixed typos
- Restructured syntax and semantics

Changes from version 05 to version 05:

- Fixed the ABNF for choices (no longer need to write a: (b/c))
- Added group choices (//)
- Added /= and //=
- Added experimental socket/plug
- Added aliases text, bytes, null to prelude
- Documented generics
- Fixed more typos

Changes from 06 to 07:

- .cbor, .cborseq, .within, .and
- Define .size on uint
- Extended Diagnostic Notation
- Precedence discussion and table
- Remove some of the "issues" that can only be understood with historical context
- Prefer "text" over "tstr" in some of the examples
- Add "unsigned" to the prelude

Appendix D. 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 / annotator) S type2]
        / "#" "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]

   type2 = value
         / typename [genericarg]
         / "(" type ")"

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

   annotator = "." 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 "=>"
             / bareword S ":"
             / value S ":"

   bareword = id

   optcom = S ["," S]

   occur = [uint] "*" [uint]
```

```
        / "+"
        / "?"

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

   value = number
         / string

   int = ["-"] uint

   ; This is a float if it has fraction or exponent; int otherwise
   number = int ["." fraction] ["e" exponent ]
   fraction = 1*DIGIT
   exponent = int

   string = %x22 *SCHAR %x22
   SCHAR = %x20-21 / %x23-7E / SESC
   SESC = "\" %x20-7E

   id = EALPHA *(*("-" / ".") (EALPHA / DIGIT))
   ALPHA = %x41-5A / %x61-7A
   EALPHA = %x41-5A / %x61-7A / "@" / "_" / "$"
   DIGIT = %x30-39
   DIGIT1 = %x31-39
   S = *WS
   WS = SP / NL
   SP = %x20
   NL = COMMENT / CRLF
   COMMENT = ";" *(SP / VCHAR) CRLF
   VCHAR = %x21-7E
   CRLF = %x0A / %x0D.0A
```

Figure 15: CDDL ABNF

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 = #6.21(any)
eb16 = #6.21(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 16: 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.

Appendix F. The CDDL tool

A rough CDDL tool is available. For CDDL specifications that do not use recursion, 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 17: CDDL tool usage

Install on a system with a modern Ruby via:

```
gem install cddl
```

Figure 18

The accompanying CBOR diagnostic tools (which are automatically installed by the above) are described in https://github.com/cabo/cbor-diag ; 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 binary strings

Examples often benefit from some white space (spaces, line breaks) in binary strings. In extended diagnostic notation, white space is ignored in prefixed binary 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 binary strings

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, binary 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 binary 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 binary strings.

G.3. Concatenated Strings

While the ability to include white space enables line-breaking of encoded binary strings, a mechanism is needed to be able to include text strings as well as binary 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 binary strings do not mix within such a concatenation, except that binary 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.4. 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.5. 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 binary 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 "#".)

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 annotation 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.

[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.

[_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.)

[_abnftodo] TO DO: This doesn't allow non-ASCII characters in the text
strings yet; there is no value notation for byte strings;
representation indicators are missing as well.

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

Authors' Addresses

Christoph Vigano
Universitaet Bremen

Email: christoph.vigano@uni-bremen.de

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
Darmstadt 64295
Germany

Email: henk.birkholz@sit.fraunhofer.de