


Featured Download

[Enterprise – Building Better Customer Experiences](#)

Exciting new System z capabilities can help organizations innovate, differentiate and deliver greater value – and do it all with a level of efficiency, economics and trust unmatched in the industry.



[Learn More](#)

[Home](#) / [Browse](#) / [YAML Ain't Markup Language](#) / [Mail](#) / [Archive](#)

YAML Ain't Markup Language

- Summary
- Files
- Reviews
- Support
- Develop
- Tracker
- Mailing Lists
- Forums
- Code

Email Archive: [yaml-core](#) (read-only)

2001:	Jan	Feb	Mar	Apr	May (101)	Jun (157)	Jul (89)	Aug (135)	Sep (17)	Oct (86)	Nov (410)	Dec (311)
2002:	Jan (76)	Feb (100)	Mar (139)	Apr (138)	May (234)	Jun (178)	Jul (271)	Aug (286)	Sep (816)	Oct (50)	Nov (28)	Dec (137)
2003:	Jan (62)	Feb (25)	Mar (97)	Apr (34)	May (35)	Jun (32)	Jul (32)	Aug (57)	Sep (67)	Oct (176)	Nov (36)	Dec (37)
2004:	Jan (20)	Feb (93)	Mar (16)	Apr (36)	May (59)	Jun (48)	Jul (20)	Aug (154)	Sep (868)	Oct (41)	Nov (63)	Dec (60)
2005:	Jan (59)	Feb (15)	Mar (16)	Apr (14)	May (19)	Jun (16)	Jul (25)	Aug (19)	Sep (7)	Oct (12)	Nov (18)	Dec (41)
2006:	Jan (16)	Feb (65)	Mar (51)	Apr (75)	May (38)	Jun (25)	Jul (23)	Aug (16)	Sep (24)	Oct (3)	Nov (1)	Dec (10)
2007:	Jan (4)	Feb (5)	Mar (7)	Apr (29)	May (38)	Jun (3)	Jul (1)	Aug (17)	Sep (1)	Oct	Nov (11)	Dec (16)
2008:	Jan (11)	Feb (4)	Mar (7)	Apr (48)	May (17)	Jun (9)	Jul (6)	Aug (12)	Sep (5)	Oct (7)	Nov (4)	Dec (11)
2009:	Jan (15)	Feb (28)	Mar (12)	Apr (44)	May (6)	Jun (16)	Jul (6)	Aug (37)	Sep (107)	Oct (24)	Nov (30)	Dec (22)
2010:	Jan (8)	Feb (16)	Mar (11)	Apr (28)	May (9)	Jun (26)	Jul (7)	Aug (25)	Sep (2)	Oct	Nov	Dec
2011:	Jan (5)	Feb (6)	Mar (3)	Apr (2)	May (10)	Jun (44)	Jul (11)	Aug (8)	Sep (6)	Oct (42)	Nov (19)	Dec (5)
2012:	Jan (23)	Feb (8)	Mar (9)	Apr (11)	May (2)	Jun (11)	Jul	Aug (18)	Sep (1)	Oct (15)	Nov (14)	Dec (8)
2013:	Jan (5)	Feb (13)	Mar (2)	Apr (10)	May	Jun (6)	Jul (17)	Aug (2)	Sep (2)	Oct	Nov	Dec

[Yaml-core] 24 May 2001 Working Draft

From: Clark C. Evans <cce@cl...> - 2001-05-25 00:04

After a serious amount of effort over the last week, I believe that I have created a working draft which reflects the current consensus on this list. I have posted this and have updated the web site accordingly.

<http://yaml.org/24may2001.html>

This is still a draft. It lacks the MIME productions or information on the bindings. Furthermore, I assume that many of the productions have bugs in them. It was very interesting task bringing our "fuzzy ideas" into a rather fixed form. Quite a bit of work... alot more than what I had expected. Anyway, after having done this I can say that YAML is not a simple language, however, I do believe that it strikes a nice balance of concerns. It will be interesting to hear feedback!

...

This has taken quite a large drain on my "day job" and I won't be all that active here until September due to rising financial pressures. So, I'm sorry to report that I won't be implementing it any time soon, but I believe Oren and Brian have promised to give it a shot.

Kind Regards,

Clark

Re: [Yaml-core] 24 May 2001 Working Draft

From: Clark C. Evans <cce@cl...> - 2001-05-25 00:14

On Thu, May 24, 2001 at 08:05:33PM -0500, Clark C. Evans wrote:
| <http://yaml.org/24may2001.html>

For discussion purposes here is a plain text version.
It's not all that pretty since the formatting is stripped.
However it should help discussion.

...

Yet Another Markup Language (YAML) 1.0

Working Draft 24 May 2001

This version:

<http://yaml.org/24may2001.html>

Editors:

Brian Ingerson <briani@...>
Clark Evans, Xgenda, Inc. <cce+yaml@...>
Oren Ben-Kiki <orenbk@...>

Copyright © 2000 Clark Evans, All Rights Reserved. This document may be freely copied provided it is not modified.

Abstract

YAML (pronounced "yaamel") is a straight-forward machine parable data serialization format designed for human readability and interaction with scripting languages such as Perl and Python. YAML is optimized for configuration settings, log files, Internet messaging and filtering. This specification describes the serialization format, a "C" API for the parser and emitter, and several language bindings.

Status of this Document

This specification is a working draft and reflects consensus reached by the members of the yaml-core mailing list. Any questions regarding this draft should be raised on this list. This is a draft and changes are expected, therefore implementers should closely follow this mailing list to stay up-to-date on trends and announcements.

There are many productions which may have bugs, and the MIME productions are yet to be completed, although this should not stop an implementer! Feedback is welcome.

Table of Contents

- 1 Introduction
 - 1.1 Origin and Goals
 - 1.2 Key Concepts
 - 1.3 Example
 - 1.4 Relation to XML
 - 1.5 Terminology
- 2 Serialization
 - 2.1 Information Model
 - 2.2 Characters
 - 2.2.1 Character Set
 - 2.2.2 Encoding
 - 2.2.3 End-of-Line Normalization
 - 2.2.4 Indicators
 - 2.2.5 Escape Sequences
 - 2.2.6 Miscellaneous Characters
 - 2.3 Strings
 - 2.3.1 Indentation
 - 2.3.2 Whitespace Folding
 - 2.3.3 Quoted String
 - 2.3.4 Simple String
 - 2.3.5 Anchor String
 - 2.3.6 Domain String
 - 2.3.7 Class String
 - 2.4 Document
 - 2.4.1 In-Line
 - 2.4.2 Node
 - 2.4.3 Classes
 - 2.4.4 Reference
 - 2.4.5 List
 - 2.4.6 Map
 - 2.5 Scalar
 - 2.5.1 Simple Scalar
 - 2.5.2 Quoted
 - 2.5.3 Block Scalar
 - 2.5.4 Multiline Scalar
 - 2.5.5 Raw Scalar
 - 2.6 MIME
- 3 Sequential Processing
 - 3.1 Parser Interface
 - 3.2 Emitter Interface
 - 3.3 Interface Converter
- 4 Language Bindings
 - 4.1 "C" Language
 - 4.2 Python
 - 4.3 Perl
 - 4.4 Java

1 Introduction

Yet Another Markup Language, abbreviated YAML, describes a class of data objects called YAML documents and partially describes the behavior of computer programs that process them.

YAML documents encode into a serialized form information having a recursive scalar, map, or list structure. YAML also includes a method to encode references and binary values. At its core, a YAML document consists of a sequence of characters, some of which are considered part of the document's content, and others that are used to indicate structure within the information stream.

A software module called a YAML parser is used to read YAML documents and provide access to their content and structure. In a similar way, a YAML emitter is used to write YAML documents, serializing their content and structure. A YAML processor is a module that provides for parser or emitter functionality or both. It is assumed that a YAML processor does its work on behalf of another module, called an application. This specification describes the interface and required

behavior of an YAML processor in terms of how it must read or write YAML documents and the information it must provide or obtain from the application.

1.1 Origin and Goals

The need for a human readable, machine parse-able graph serialization format was conceived independently by two parties. Brian Ingerson had developed software for the storage of native Perl data structures called `Data::Denter`. And Clark Evans was looking for a minimal and easy to use data serialization language for C++ and Python applications. Shortly after Clark Evans first proposed YAML, Brian's work on `Data::Denter` was brought to his attention. YAML is the result is a unification of each individuals work.

The design goals for YAML are:

1. YAML documents are very readable by humans.
2. YAML interacts well with scripting languages.
3. YAML uses host language's native data structures.
4. YAML works well with Internet mail architecture
5. YAML allows binary and large formatted text.
6. YAML has a consistent information model.
7. YAML includes a stream based interface.
8. YAML is expressive and extensible.
9. YAML is easy to implement.

YAML was designed with experience gained from the construction and deployment of `Data::Denter`. YAML has also enjoyed much markup language critique from SML-DEV list participants, including experience with the Minimal XML and Common XML specifications.

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 822 for mail headers and Internet RFC 2045/6 for MIME sections), provides all the information necessary to understand YAML Version 1.0 and construct computer programs to process it.

1.2 Key Concepts

YAML builds upon the structures and concepts described by XML, SOAP, Perl, HTML, Python, C, rfc0822, rfc2046, and SAX.

YAML uses similar type structure as Perl. In YAML, there are three fundamental structures: scalars, maps (`%`) and lists (`@`). YAML also supports references to enable the serialization of graphs. Furthermore, each data value can be associated with a class name to allow the use of specific data types. This type structure common to many other languages and provides a solid basis for an information model. Furthermore, it enables the programmer to use their programming language's native data constructs for YAML manipulation, instead of a document object.

YAML uses similar whitespace handling as HTML. In YAML, sequences of spaces, tabs, and carriage return characters are usually folded into a single space during parse. This wonderful technique makes markup code readable by enabling indentation and word-wrapping without affecting the canonical form of the content. This folding technique can also be found in the structured headers of RFC 822.

YAML uses block scoping similar to Python. In YAML, the extent of a node is indicated by its child's nesting level, i.e., what column it is in. Block indenting provides for easy inspection of the document's structure and greatly improves readability. This block scoping is possible due to the whitespace folding technique.

YAML uses quoted strings similar to C. In YAML, scalars can be surrounded by quotes enabling escape sequences such as `\n` to represent a new line, `\t` to represent a tab, and `\\` to represent the slash. Unlike C, since whitespace is folded, YAML introduces bash style `"\"` to escape additional spaces that are part of the content and should not be folded. Further, the trailing `\` is used as a continuation marker, allowing content to be broken into multiple lines without introducing unwanted whitespace. Lastly, ISO 8859-1 characters can be specified using `"\x3B"` style escapes and UNICODE characters can be specified using a similar technique `"\u003B"`.

YAML allows for a rfc822 compatible header area to enable direct usage in mail handlers. These headers also provide a place for comments, specific processing instructions, and encoding declarations. This provides a flexible and forward looking method to

augment the YAML parser with other features such as a validator similar to TREX or RELAX should the become necessary.

YAML supports binary and formatted text entities with MIME multi-part attachments. Each attachment is given an reference identifier that can be associated with a location in hierarchical YAML content. This allows leaf values that would disrupt the in-line structural flow to be handled out of band in a separate block mechanism.

In addition to a native in-memory load/save interface, YAML provides both a pull style input stream and a push style (SAX like) output stream interface. This enables YAML to directly support the processing of large documents, such as a transaction log, or continuous streams, such as a feed from a production machine.

1.3 Example

Following is a simple example of an invoice represented as a YAML

document. The colon is used to separate name value pairs. The percent sign following a colon indicates that the value is an mapping. The at sign indicates that the value is an ordered list.

```
buyer: %
  address      : %
    city       : Royal Oak
    line one   : 458 Wittigen's Way
    line two   : Suite #292
    postal     : 48046
    state      : MI
  family name  : Dumars
  given name   : Chris
date          : 12-JAN-2001
delivery: %
  method      : UZS Express Overnight
  price       : $45.50
comments :
  Mr. Dumars is frequently gone in the morning
  so it is best advised to try things in late
  afternoon. If Joe isn't around, try his house
  keeper, Nancy Billsmer @ (734) 338-4338.
invoice      : 00034843
product : @
  %
    desc      : Grade A, Leather Hide Basketball
    id        : BL394D
    price     : $450.00
    quantity  : 4
  %
    desc      : Super Hoop (tm)
    id        : BL4438H
    price     : $2,392.00
    quantity  : 1
tax          : $0.00
total       : $4237.50
```

1.4 Relation to XML

There are many differences between YAML and the eXtensible Markup Language ("XML"). XML was designed to be backwards compatible with Standard Generalized Markup Language ("SGML") and thus had many design constraints placed on it that YAML does not share. Also XML, inheriting SGML's legacy, is designed to support structured documents, where YAML is more closely targeted at messaging with direct support for the native data structures of modern programming languages. Further, XML is a pioneer in many domains and YAML has been grown on the lessons learned by the XML community. These points aside, there are many differences.

The YAML and XML information model are starkly different. In XML, the primary construct is an attributed tree, with where each element has a ordered, named list of children and an unordered mapping of names to strings. In YAML, the primary hierarchical construct alternates between an anonymous list, named map, and scalar values. This difference is critical since YAML's model is directly supported by native data structures in most modern programming languages, where

XML's model requires mapping, conventions, or an alternative programming component, a document object model.

The YAML and XML syntax vary significantly. In XML, tags are used to denote the begin and end of an element. In YAML, scope is designated by indentation. Where YAML builds upon RFC 822 and MIME for it's

declaration section and support for binary and/or large scalar values, XML has it's own processing instruction mechanism and relies upon layered technologies for support for binary leaves. Furthermore, YAML has a simple whitespace policy, where XML's whitespace policy is completely configurable.

1.5 Terminology

The terminology used to describe YAML is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of a YAML processor:

may

Conformant YAML texts and processors are permitted to but need not behave as described.

must

Conformant YAML texts and processors are required to behave as described, otherwise they are in error.

error

A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.

fatal error

An error which a conforming YAML processor must detect and report to the application. After encountering a fatal error, the processor may continue processing the data to search for further error to report to the application.

2 Serialization

A YAML document can reside in many different forms as long as it complies with the information model below. This includes a sequence of bytes in memory, on disk, or arriving via a network socket as much as it includes events from a sequential interface or an language specific in-memory representation. After covering the information model, this section focuses on the serialized representation

2.1 Information Model

The information model for YAML is a directed acyclic graph having map, list, and scalar nodes. These data structures are directly supported in modern programming languages such as Python, Perl, Java, and C++.

Since the information model is a graph, a separate serialization model is required. The serialization model adds an additional node type, the reference, to record subsequent occurrences of a given node within a sequence. This enables a more compact notation so that duplicate occurrences of a given node need not be serialized more than once.

document A YAML document is a ordered sequence of one or more nodes viewed as a stack, where the most recent node to be added to the sequence is at the top of the stack. At the bottom of the stack is a mandatory header node.

list An ordered sequence of zero or more nodes. Nodes are included by reference, thus they may be part of more than one list or map.

node A YAML node can be one of three types: list, map, scalar. Every node may optionally have a class.

map An un-ordered sequence of zero or more (key, node) tuples where the key is unique within the sequence.

scalar An ordered sequence of zero or more bits. Where bits are an atomic construct able to have a value of one or zero.

key A sequence of zero or more characters.

class An opaque object that has a serialization which complies with the class production.

header This is a special node used for configuration settings. It is restricted to be a map of scalars (key,scalar) where each scalar is limited to the character production. When serialized, this header may be implicit. See the header section for details.

The serialization model adds an anchor attribute to every node, and introduces the reference node. The reference node advertises an anchor to indicate the repetition of a node previously encountered.

anchor An additional attribute added to each node that provides for identification of the node within a given node sequence. Only nodes which could be referenced further in the sequence must be given an anchor. It is not necessary that the anchor is unique.

reference An additional node type consisting of an anchor which is

used to indicate the repetition of the node with the same anchor most previously encountered.

In the serialization model, it is important that each node is serialized exactly once. If a node appears more than once in the graph, only the first occurrence of the node should be serialized. All remaining occurrences of this node should be represented with reference nodes. In this scheme, if a YAML document is loaded into an random access representation, then the reference nodes and anchor indicators should not be available as the non-serialized information model should be used. Also note that anchors can repeat to allow for concatenation, although only the most recent node with a given anchor may be referenced.

2.2 Characters

Characters are the basis for a serialized version of a YAML document. Below is a general definition of a character followed by several characters which have specific meaning in particular contexts.

2.2.1 Character Set

By default, serialized YAML uses a subset of the ISO/IEC 10646 character set. Specifically, YAML uses those characters expressible as a single 16 bit value with the UTF-16 character encoding.

```
[01] char  ::  #x9 | #xA | [#x20-#xD7FF] | /* A single Unicode
              =  [#xE000-#xFFFD]          character representable
                                           by a 16 bit value,
                                           including the space and
                                           new line characters */
```

Due to the end of line normalization rules, the carriage return (#xD) is not included. As with standard practice, the the surrogate block, FFFE and FFFF are excluded. Further, the range [#x10000-#x10FFFF] is excluded since it would require compliant YAML processors to be concerned with surrogate pair handling.

In the information model scalar values may be binary and may not comply within the above character set. In this case, the scalar must be serialized using MIME section, perhaps using a base64 transfer encoding.

2.2.2 Encoding

A YAML processor is required to support both UTF-16 and UTF-8 character encodings. If an input stream begins with a byte order mark, then the initial character encoding shall be UTF-16. Otherwise, the initial encoding shall be UTF-8.

```
[02] bom   ::  #xFEFF /* The Unicode ZERO WIDTH NON-BREAKING SPACE
              =          character used to mark a UTF-16 stream and
                        determine byte ordering. */
```

If the stream begins with #xFFFE, then the byte order of the input stream must be swapped when reading.

2.2.3 End-of-Line Normalization

On input and before parsing, a compliant YAML parser must translate both the two-character sequence #xD #xA (CR LF) and any #xD (CR) which is not followed by a #xA (LF) into a single #xA (LF) character. This allows for the definition of an end-of-line marker.

```
[03] eol   ::=  #xA          /* a normalized end of line marker */
[04] blank ::=  eol eol      /* a blank line */
```

On output, a YAML emitter is free to serialize end of line markers using what ever convention is most appropriate. For Internet mail, CRLF is the preferred form.

2.2.4 Indicators

Indicators are special characters which are used to describe the structure of a YAML document.

```
[05] imap   ::=  '%' /* indicates a map node */
[06] ilist  ::=  '@' /* indicates a list node */
[07] iquote ::=  '"' /* indicates a quoted string */
[08] ikey   ::=  ':' /* separates a key from a node */
[09] iblock ::=  '|' /* indicates a block scalar */
```

```

[10] iblockend ::= '\ ' /* indicates end of a block */
[11] iescape   ::= '\ ' /* indicates an escape sequence */
[12] iref      ::= '*' /* indicates a reference node */

[13] iclass    ::= '#' /* indicates a class attribute */
[14] inull     ::= '~' /* indicates a null node */
[15] ianchor   ::= '&' /* indicates an anchor attribute */
[16] ireserved :: '^' | '!' | '`' | ',' | ';' | '.' /*
= reserved *
/
[17] indicator :: imap | ilist | iquote | ikey | /*
= iblock | iblockend | iescape | indicator
iref | iclass | ianchor | ireserved characters
*/

```

2.2.5 Escape Sequences

Escape sequences are used to denote significant whitespace, specify characters by a hexadecimal value, and produce the literal quote and escape indicators.

```

[18] eescape :: iescape iescape /* escape
= literal */
[19] equote  :: iescape iquote /* quote
= literal */
[20] eeol    :: iescape 'n' /* new line *
= /
[21] etab    :: iescape 't' /* tab */
=
[22] espace  :: iescape #x20 /* space */
=
[23] ex2     :: iescape 'x' hex hex /* 8 bit
= character */
[24] eu4     :: iescape 'u' hex hex hex hex /* 16 bit
= character */
[25] escape  :: eescape | equote | eeol | etab | /* escape
= espace | ex2 | eu4 sequences */

```

2.2.6 Miscellaneous Characters

This section includes several common character range definitions.

```

[26] lwsp    :: #x20 | #x9 /* Linear whitespace, the space
= or tab character. */
[27] wsp     :: lwsp | eol /* A single whitespace
= character, including the space,
tab, new line. Excluding the
carriage return. */
[28] pchr    :: char - wsp /* Non-whitespace characters */
=
[29] qchar    :: ( ( pchr /* printables less the quote and
= - iquote ) escape character. */
- iescape )
[30] ichr     :: ( pchr /* printables less the indicator
= - indicator ) characters. */
[31] alpha    :: [#x41-#x5a] | /* ASCII alphabetic character,
= [#x61-#x7a] a-z and A-Z. */
[32] number   :: [#x0030-#x0039] /* ASCII numeric character, 0-9
= */
[33] alphanum :: alpha | number /* ASCII alpha numeric character
= */
[34] hex      :: number | /* one hexadecimal digit 0-9
= [#x0041-#x0046] or A-F */

```

2.3 Strings

Moving on to a higher level of abstraction, are sequences of characters, or strings. This section describes whitespace folding and indentation policies, as well as quoted and raw strings.

2.3.1 Indentation

In a YAML serialization, structure is determined from indentation, where indentation is defined as an end of line marker followed by zero or more spaces or tabs. Indentation level is defined recursively.

```

[35] tab(0)  :: eol /* The first level of indentation
= is a single end of line marker
(new line). */
[36] tab(n)  :: tab(n-1) lwsp+ /* The previous indentation
= setting plus one or more spaces or

```



```

        leading plus one or more spaces or
        tabs. */

```

It should be noted that this production does not imply that `tab(3)` is a constant throughout the entire serialization, as the particular indentation style may change over time. Specifically, an indentation level `tab(n)` is set with the first subordinate line of a node at `tab(n-1)`. With the exception of quoted strings, all subordinate lines must share this same indentation setting. This allows indentation to vary over the serialization as long as it remains constant for a given node's content.

2.3.2 Whitespace Folding

Since a YAML serialization uses whitespace for indentation, in many cases the parser must condense sequences of one or more whitespace characters into a single space (`#x20`). When this functionality is implied, the fold production below will be used.

```

[37] fold    ::=  wsp+      /* folded whitespace */
[38] lfold   ::=  lws+     /* linear folded whitespace */

```

2.3.3 Quoted String

A quoted string is a mechanism to treat a sequence of characters as a single unit. Within a quoted string, indicators (with the exception of `"` and `\`) can be used without worry, indentation rules are suspended, and escape sequences can be used to introduce significant whitespace which would otherwise be subject to folding.

A quoted string begins and ends with a quote character. It can extend for as many lines as necessary, although an editor or emitter is free to re-break and indent a quoted string as needed to maintain readability. A quoted string cannot contain an un-escaped quote or an invalid escape sequence.

```

[39] qstr    ::=  iquote ( qchar | fold | escape ) *    /* quoted
                =  iquote                                string */

```

2.3.4 Simple String

There are two sorts of simple strings, ones that are subject to the fold technique and ones that are not. We call the first type of string folded and the second type raw.

```

[40] rstr    ::=  ( pchr | lws ) *                      /* a raw single line
                =  string */
[41] fstr    ::=  pchr+ ( lfold pchr+ ) *               /* folded string */
                =
[42] fistr   ::=  ichr+ ( lfold ichr+ ) *               /* indicator free, folded
                =  string */

```

A folded string starts with a printable which is not an indicator. It is then followed by a sequence of printable words with intermediate folded whitespace. At parse time, the folded space must be condensed into a single space (`#x20`) character.

2.3.5 Anchor String

An anchor string is a sequence of numeric digits used when referencing a node previously visited in the document stream.

```

[43] astr    ::=  number+    /* Any sequence of digits 0-9. */

```

2.3.6 Domain String

There are several strings of interest when dealing with domain names. They are included below.

```

[44] dnsrev   ::=  dnstop ( '.' dnsseg ) *              /* A reverse dns name *
                =  /
[45] dnstop   ::=  alpha | ( alpha                      /* A top level domain,
                =  ( alphanum | '-' ) *                  like "com", "org", ...
                alphanum )                               */
[46] dnsseg   ::=  alphanum | ( alphanum               /* A domain name
                =  ( alphanum | '-' ) *                  segment */
                alphanum )

```

2.3.7 Class String

Each node can be adorned with a class string. There are three types of class strings: local, qualified, and built-in.

```
[47] cstr  :: cloc | cqual |      /* A local, qualified or built-in
      =  cbltn                class name. */
```

Local class strings, which are unique within a controlled context, begin with an underscore. These strings are not globally unique and should only be used for local usage or for experimentation where data exchange is not likely.

```
[48] cloc  ::= '_' dnsrev  /* A local class string */
```

Qualified class strings are globally unique since they are serialized with a reverse domain name format similar to

"com.clarkevans.timesheet", according to the class production. The definition of a qualified class is determined by the domain name holder.

```
[49] cqual ::= dnsrev  /* A reverse dns qualified class string */
```

Built-in class strings, reserved for YAML usage, are also globally unique and are serialized without a period. Core classes include real, designating an imprecise floating point value, and int, designating a precise integer value. Each string using this style must be considered equivalent to the same string obtained by concatenation with "org.yaml." For example, the built-in class string "int" is considered equivalent to the qualified class string "org.yaml.int".

```
[50] cbltn ::= dnsseg  /* A built-in class name (no period) */
```

2.4 Document

A serialized object is a YAML document if, taken as a whole, it complies with the following production.

```
[51] document  :: bom? (      /* A byte order mark followed by
      =  multi-part | either a sequence of nodes (stack)
      inline )?      or a multi-part MIME sequence. */
```

2.4.1 In-Line

This form of serialization separates each top level node with a blank line. Since the YAML serialization depends upon indentation level to delineate blocks, each node production is a function of an integer, much like the tab() production above. This production is the beginning of the recursion.

```
[52] inline  :: (node(0) blank)* /* A sequence of zero or more
      =                      nodes at indentation level 0
                        separated by a blank line. */
```

TODO: Describe how the first node(0) may be a map which conforming to the header described above. Detail how if a "X-YAML-Version" entry in the first header is missing, that the header is assumed since it is mandatory in the information model. Also describe how this header, if provided, is used to determine if the multi-part style is to be used.

2.4.2 Node

A node begins at a particular level of indentation, n, and itself is indented at a level n+1. A node can either be a scalar, map, list, or reference.

```
[53] node(n)  :: ( node_color lwsp+ )? /* an optional
      =  map(n) | scalar(n) | node color
      list(n) | reference followed by either
                        a map, list,
                        scalar, or
                        reference */
[54] node_color :: anchor | class | /* either an
      =  ( anchor lwsp+ class ) | anchor, or class,
      ( class lwsp+ anchor ) or both in any
                        order */
```

2.2.3 Class

YAML indirectly supports the serialization of data types and object class name with a class attribute on each node. If this attribute is provided, then each specific language binding may use this information to dress the node appropriately, otherwise a warning must be issued and the must data treated as if the class attribute was not provided.

```
[55] class  ::  iclass      /* Associates an class string with a
      =      cstr          given node */

"This is a scalar node without a class name."
#com.clarkevans.ts  "Scalar with qualified class name."
#_local            "Scalar with local domain name."
#real              "23.42"
"The above scalar node is a built-in class name."
```

2.2.4 Reference

An anchor is an indicator which can be used to color a node giving it an sequential numeric digit for an identifier. The reference node type can then be used to indicate additional inclusions of an anchored node. The anchor string of a reference refers to the most recent node having that same anchor string. It is an error to have a reference use an anchor string which does not occur previously in the serialization.

```
[56] anchor  ::  ianchor    /* Associates an anchor string with a
      =      astr          given node for further reference */

[57] ref      ::  iref       /* A reference node */
      =      astr

"A scalar node without a reference"
&0001 "The next node is a reference to this one."
*0001
#_aclass &0001 "Node with local class and anchor."
```

2.4.5 List

A list is the simplest form of node, it is a sequence of nodes at a higher indentation.

```
[58] list(n)  ::  ilist      /* A list indicator,
      =      (tab(n) node(n+1))* followed by zero or more
                                   indented nodes. */

@
"First item in top list"
@
"Subordinate list"
@
"Above list is empty"
"Fifth item in top list"
```

2.4.6 Map

A map is a unique association of keys with values. Where a key is either a quoted or folded string.

```
[59] map(n)   ::  imap ( tab(n)      /* A map indicator, followed
      =      key lws+ ikey          by a sequence of key/node
      mnd(n+1) ) *                  pairs indented
                                   appropriately. Either a
                                   space, or the appropriate
                                   indentation precedes */

[60] key       ::  fistr | qstr      /* a single line folded
      =                                     string, or a quoted string */
                                   /

[61] mnd(i)    ::  (lws node(i)) |    /* node, including the
      =      ((lws node_color)?      multiline, with map specific
      ( tab(i) | lws )               whitespace handling options.
      ( block(i) |                   */
      multiline(i) ))
```

In a given map, there is the further restriction that within a given map, two folded key values cannot be identical.

```
%
first : "First entry"
second: %
      key: "Subordinate map!"
third item: @
      "Subordinate list"
      %
      "Previous map is empty."
"@": "This key had to be quoted."
```

2.5 Scalars

The while most of the document productions above are fairly strict, the scalar production is generous. It offers four ways to express scalar values depending upon the type of value. The table below describes the various styles.

	Indented?	Folded?	Escaped?	Indicators?	In-line
Quoted	No	Yes	Yes	Yes	Yes
Block	Yes	No	No	Yes	Yes
Simple	No	Yes	No	No	Yes
Multiline	Yes	No	No	Yes	No
Raw	No	No	No	Yes	No

The following scalar forms can be used in lists, blocks, and as part of the top level sequence

```
[62] scalar(n)  :: simple(n) | block(n) |      /* in-line scalar
               =   quote(n)                nodes */
```

2.5.1 Simple Scalar

In this simple one line case, the scalar is folded may not contain

indicators and is not escaped.

but it may not contain indicators

```
[63] simple(n)  :: ichr fstr? /* a single non-indented,
               =               non-indicator, non-escaped, folded
                               line */
```

This is a simple, one line scalar.

```
@
  This   intermediate space is not significant.
  This has an an intermediate @ indicator character.
%
  key: They may be used within a map.
```

2.5.2 Quoted Scalar

A quoted scalar uses quoted strings. With this mechanism, the indicators (excepting the quote and slash) can be used freely, whitespace is folded, and escape sequences can be used to indicate significant whitespace. Since a quoted string has an end marker, indenting rules are suspended for its content allowing for easy cut and paste, although quotes, significant whitespace, and slashes must still be escaped.

```
[64] quoted(n) ::= qstr /* A quoted scalar value. */
```

"Quoted scalar with two lines.\nWith a new line."

```
@
  "Spaces are folded so this
  new line is not significant."
  "Furthermore indicators such
  as @ # : can be added."
  "Escape sequences can be used
  to removed or \ add white\
  space, \\ and \" must be escaped."
  "These quote lines may also
break indentation."
```

2.5.2 Block Scalar

This mechanism supports source code or other scalar values with significant use of indicators, quoted escaping, or significant whitespace. To retain readability for this case, the block scalar allows raw text strings to be used in an indented fashion where each line indented is preceded by a block indicator. All whitespace is significant up to and including the end of line marker. To end the block, a block end indicator is used and any characters on this line are also included, up to and not including the end of line marker.

```
[65] block(n)  :: ( iblock rstr tab(n) ) * /* An indented
               =   iblockend rstr          character block */
```

```
|This is a block scalar,   with significant
|whitespace, and the use of " @, etc.
|The \ is used to signify the end of the block.
\
```

```
%
  first: |
```

```

    |Has a leading and trailing new line.
    \
second:
    |No leading, nor does this have a
    \trailing new line.
third:
    | leading spaces are significant
    \

```

2.5.4 Multiline Scalar

In this is a special case only usable by maps, is very similar to the simple case only that it allows for multi-line content. IN this case, each line is indented, whitespace is folded, indicators are forbidden and escaping is not available. To delineate the end of this simple scalar, indentation is employed.

```

[66] multiline(n)  ::  ichr ( fstr? tab(n) ) *    /* one or more
                    =    fstr                      indented,
                                                non-escaped, folded
                                                characters */

```

```

%
first: This is a multi line scalar without a
      leading indicator or significant whitespace.
second:
  A multi-line scalar can begin on
  the second line.
third:
  <html><head><title>Embedded HTML!</title></head>
  <body><p class="none">This can even have embedded
  HTML since there is no escaping, and html begins
  with < which is not an indicator!
  </p></body>

```

2.5.5 Raw Scalar

The raw scalar is only applicable within a MIME section. The sequence of strings does shall not contain the end marker.

```

[67] raw(end)    ::  ( (rstr - end ) eol )      /* A sequence of raw
                    =    (rstr - end )?          strings, where each
                                                string does not contain
                                                the end marker */

```

```

XX XXX    XXXXX  XX  XX
XXX XX      X  XX X XX
XX    XXXXXX  XX X XX
XX    X  XX  XXXXXXX
XXXX    XXXXX X  XX XX

```

2.6 MIME

Instead of an in-line form, an rfc2046 like multi-part form can be given. A YAML processor must implement both quoted printable and base64 encodings and must support this multi-part form. The header, and then each subsequent section is treated as a node in the top-level document stack.

The following scalar forms can be used in lists, blocks, and as part of the top level sequence

```

[68] multipart  ::  TODO: Describe              /* the multi-part
                    =    multi-part productions consists of a single
                                                in-line section followed
                                                by one or more parts. */

```

TODO: This section will describe the productions necessary to be compatible with MIME and offer raw scalar values with possibly different character encodings, or a transfer encoding of either binary, base64, or quoted printable.

NOTE: This may look like a great deal of work, but in general, it shouldn't be that bad at all. Specifically, it makes YAML mail system ready.

```
Date: Sun, 13 May 2001 23:48:04 -0400
MIME-Version: 1.0
Content-Type: multipart/related;
    boundary="=====
X-YAML-Version: 1.0
```

```
-----
Content-Type: text/plain; id="0001"
```

```

XX  XXX      XXXXX      XX   XX
  XXX XX      X      XX X XX
XX      XXXXXX      XX X XX
XX      X   XX      XXXXXXXX
XXXX      XXXXXX X      XX XX

```

```
-----
Content-Type: image/gif; id="0002"
Content-Transfer-Encoding: base64
```

R0LGODlh7+/GbOzAJmZAIEHGBmZGbmZgBmZgBmZgBmZgBmZgBmZgBmZgBmZgBmZCH+EKAAYALAAAAAAZAA8AQAR70EgZArI BWHwNts1gB6R
GV0wCBMlwrFI4VEulSMYrLChhyRYLq4MDKYrm9XuFQuIZLhALA
pm6VV+g412eBa4jcWZ3gHeCJQJycXSJEzaIc5SIwZ0RADs=

```
-----
Content-Type: text/x-yaml; id="0003"
```

```

an inline : &0004
    This is a folded text entity
    that is associated with a
    reference.
content :
    comment:
        The cyclic item is a reference
        to the top-level map.
cyclic : *0003
image : *0002
inline : *0004
raw : *0001
title : This contains multiple references

```

3 Sequential Processing

There are two basic types of interfaces used for sequential processing, pull and push. The difference lies primarily in who has the primary control of the process, the consumer or the producer. With a push interface, the producer has the primary control and with a pull interface is is the consumer who drives the process. For the most control over the process, YAML's parser has a pull interface and YAML's emitter has a push interface. This enables the application which pulls from the parser and pushes to the emitter to have the primary control of the process.

3.1 Parser Interface

The parser or iterator interface provides a sequential access to a YAML document where the flow control is dictated by the consumer. The parser interface uses the serialization model, handling duplicate nodes within the graph through a reference. The parser interface has several components.

<code>cursor</code>	A structure is used to hold the current node, including the node's type (scalar, list, map, or reference), the node's class, the node's anchor, and, if the node is part of a map, the key associated with the node.
<code>open()</code>	A function which returns the document's top level list cursor. This method may require an input stream such as a string buffer, a file object, or an open socket.
<code>first()</code>	A function which takes a map or list cursor and returns a cursor for the first child within the sequence, if any. It is an error to call this function on cursor currently holding a reference or scalar.
<code>next()</code>	A method which advances the cursor passed to the next node within the current map or list. If there is not a subsequent node within the sequence, then this function must free the cursor and should indicate that additional siblings are not to be found.

`close()` is a method on the `top_level_cursor` which can be used on the

```

close()  A method on the top level cursor which can be used on the
         top level cursor to stop iteration and invalidate all
         subordinate cursors.

read()   A method which can be called on a scalar cursor to read an
         arbitrary sized chunk of it's value. A scalar's value can
         only be read once, but multiple calls to this function may
         be required.

deref()  An optional method which can be called on a cursor with a
         reference node to return a new cursor for the node
         referenced. This method need only be supported by those
         iterators over a random access input stream.

```

As part of this interface, each cursor returned from the parser must remain valid until `close()` is called, or the `next()` is called on the cursor, or `next()` is called on an ancestral cursor. Note that this implies that N cursors may be open at any given time, one cursor for each map or list in a given lineage. It is the parser's task to make sure that each cursor points to valid information.

3.2 Emitter Interface

The emitter or visitor interface interface provides a sequential access to a YAML document where the flow control is dictated by the producer. The emitter interface uses the serialization model, handling duplicate nodes within the graph through a reference. The emitter interface has several components.

```

cursor    The emitter may require a cursor for its bookkeeping which
          is opaque.

start()   A function which must be called to start the emitter. This
          function returns a cursor to be used for subsequent
          operations.

begin()   A function to be called when a map, list, or scalar node is
          to be started. This function is passed a the current cursor
          and returns a cursor to be used for subordinate operations.
          This cursor takes the node's type, a class, an anchor, and
          possibly a key if the node is in the context of a map.

write()   A method which can be called multiple times within the
          context of a scalar to provide the scalar's character
          value.

ref()     An optional method which can be called on a map or list
          cursor to add a reference to the sequence of children. This
          method takes the current cursor and an anchor as
          parameters. If this method is not provided, the caller must
          recursively visit the referenced node.

end()     A method called on a cursor to indicate that the node in
          question is no longer in scope. The cursor passed should no
          longer be used.

finish()  A method which must be called on the top level cursor to
          indicate that the document stream has finished.

```

As part of this interface, a cursor is deemed valid untill `end()` has been called on it. It is an error to call `end()` on a cursor when a subordinate cursor is still valid. This implies that the emitter need not use N cursors, and that one cursor which is re-used with a depth indicator will suffice.

3.3 Interface Converter

In some cases it is necessary to convert a pull to a push interface, or a push to a pull interface. While converting from pull to push is relatively easy, converting from push to pull requires a multi-threaded approach or a large intermediate memory buffer. To help automate conversions, the YAML processing library includes both interface converters.

4 Language Bindings

The realization of the information model depends upon the programming language. Included in this specification are the official bindings for Python, Perl, and C. A language binding is not required to implement the sequential interface.

TODO: This section needs much work

4.1 "C" Language

Since the "C" programming language does not have dynamic lists or maps, this binding only implements the sequential interface.

4.2 Python

For the Python binding, unless specified otherwise by a class, scalar values are bound to the string type, lists are bound to the list type, and maps are bound to the dictionary type with the key represented as a string.

If a scalar has a real class, then the scalar must be bound as a floating point value if possible, otherwise a warning must be issued. Likewise, if a scalar has a int class, then the scalar must be bound to a normal integer, otherwise a warning must be issued. Conversions can fail if the scalar value is not parse-able or if the size of the parsed value will not fit within the standard type.

If any other class is encountered, then the results depend upon the type. First, the class identifier is used to construct a class with the same package name. And then, depending upon the type, initialization is carried out differently. If the node is a map, then the keys are assumed to be attributes, and the setattr method is used to initialize the object. If the node is a scalar, then repr is used. Otherwise, if the node is a array, then additem method is used.

4.3 Perl

For the Perl binding...

4.4 Java

For the Java binding...

For more information on the Byte Order Mark see the Unicode FAQ.

Re: [Yaml-core] 24 May 2001 Working Draft

From: Clark C. Evans <cce@cl...> - 2001-05-25 02:20

The starting production was wrong in the initial version of this. It is now fixed.

Clark

SourceForge

About
Site Status
@sfnet_ops

Find and Develop Software

Create a Project
Software Directory
Top Downloaded Projects

Community

Blog
@sourceforge
Job Board
Resources

Help

Site Documentation
Support Request
Real-Time Support