YAML Ain't Markup Language (YAML™) Version 1.1

Working Draft 2004-12-28

Oren Ben-Kiki <oren@ben-kiki.org>
Clark Evans <cce@clarkevans.com>
Brian Ingerson <ingy@ttul.org>

YAML Ain't Markup Language (YAML™) Version 1.1
Working Draft 2004-12-28

by Oren Ben-Kiki, Clark Evans, and Brian Ingerson

Status of this Document

This specification is a draft reflecting consensus reached by members of the yaml-core mailing list [http://lists.sourceforge.net/lists/listinfo/yaml-core]. Any questions regarding this draft should be raised on this list. We expect all further changes will be strictly limited to wording corrections and fixing production bugs.

We wish to thank implementers who have tirelessly tracked earlier versions of this specification, and our fabulous user community whose feedback has both validated and clarified our direction.

Abstract

YAML™ (rhymes with “camel”) is a human-friendly, cross language, Unicode based data serialization language designed around the common native data structures of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing. Together with the Unicode standard for characters [http://www.unicode.org/], this specification provides all the information necessary to understand YAML Version 1.1 and to creating programs that process YAML information.

Table of Contents

Chapter 1. Introduction

“YAML Ain't Markup Language” (abbreviated YAML) is a data serialization language designed to be human-friendly and work well with modern programming languages for common everyday tasks. This specification is both an introduction to the YAML language and the concepts supporting it and also a complete reference of the information needed to develop applications for processing YAML.

Open, interoperable and readily understandable tools have advanced computing immensely. YAML was designed from the start to be useful and friendly to people working with data. It uses Unicode printable characters, some of which provide structural information and the rest containing the data itself. YAML achieves a unique cleanness by minimizing the amount of structural characters, and allowing the data to show itself in a natural and meaningful way. For example, indentation may be used for structure, colons separate mapping key: value pairs, and dashes are used to “bullet” lists.

There are myriad flavors of data structures, but they can all be adequately represented with three basic primitives: mappings (hashes/dictionaries), sequences (arrays/lists) and scalars (strings/numbers). YAML leverages these primitives and adds a simple typing system and aliasing mechanism to form a complete language for serializing any data structure. While most programming languages can use YAML for data serialization, YAML excels in those languages that are fundamentally built around the three basic primitives. These include the new wave of agile languages such as Perl, Python, PHP, Ruby and Javascript.

There are hundreds of different languages for programming, but only a handful of languages for storing and transferring data. Even though its potential is virtually boundless, YAML was specifically created to work well for common use cases such as: configuration files, log files, interprocess messaging, cross-language data sharing, object persistence and debugging of complex data structures. When data is easy to view and understand, programming becomes a simpler task.

1.1. Goals

The design goals for YAML are:

1. YAML is easily readable by humans.
2. YAML matches the native data structures of agile languages.
3. YAML data is portable between programming languages.
4. YAML has a consistent model to support generic tools.
5. YAML supports one-pass processing.
6. YAML is expressive and extensible.
7. YAML is easy to implement and use.

1.2. Prior Art

YAML's initial direction was set by the data serialization and markup language discussions among SML-DEV members [http://www.docuverse.com/smldev/]. Later on it directly incorporated experience from Brian Ingerson's Perl module Data::Denter [http://search.cpan.org/doc/INGY/Data-Denter-0.13/Denter.pod]. Since then YAML has matured through ideas and support from its user community.

YAML integrates and builds upon concepts described by C [http://cm.bell-labs.com/cm/cs/cbook/index.html], Java [http://java.sun.com/], Perl [http://www.perl.org/], Python [http://www.python.org/], Ruby [http://www.ruby-lang.org/], RFC0822 [http://www.ietf.org/rfc/rfc0822.txt] (MAIL), RFC1866 [http://www.ics.uci.edu/pub/ietf/html/rfc1866.txt]

(HTML), RFC2045 [http://www.ietf.org/rfc/rfc2045.txt] (MIME), RFC2396 [http://www.ietf.org/rfc/rfc2396.txt] (URI), XML [http://www.w3.org/TR/REC-xml.html], SAX [http://www.saxproject.org/] and SOAP [http://www.w3.org/TR/SOAP].

The syntax of YAML was motivated by Internet Mail (RFC0822) and remains partially compatible with that standard. Further, borrowing from MIME (RFC2045), YAML's top-level production is a stream of independent documents; ideal for message-based distributed processing systems.

YAML's indentation based scoping is similar to Python's (without the ambiguities caused by tabs). Indented blocks facilitate easy inspection of the data's structure. YAML's literal style leverages this by enabling formatted text to be cleanly mixed within an indented structure without troublesome escaping. YAML also allows the use of traditional indicator-based scoping similar to Perl's. Such flow content can be freely nested inside indented blocks.

YAML's double quoted style uses familiar C-style escape sequences. This enables ASCII encoding of non-printable or 8-bit (ISO 8859-1) characters such as “**`\x3B`**”. Non-printable 16-bit Unicode and 32-bit (ISO/IEC 10646) characters are supported with escape sequences such as “**`\u003B`**” and “**`\U0000003B`**”.

Motivated by HTML's end-of-line normalization, YAML's line folding employs an intuitive method of handling line breaks. A single line break is folded into a single space, while empty lines are interpreted as line break characters. This technique allows for paragraphs to be word-wrapped without affecting the canonical form of the content.

YAML's core type system is based on the requirements of agile languages such as Perl, Python, and Ruby. YAML directly supports both collection (mapping, sequence) and scalar content. Support for common types enables programmers to use their language's native data structures for YAML manipulation, instead of requiring a special document object model (DOM).

Like XML's SOAP, YAML supports serializing native graph data structures through an aliasing mechanism. Also like SOAP, YAML provides for application-defined types. This allows YAML to represent rich data structures required for modern distributed computing. YAML provides globally unique type names using a namespace mechanism inspired by Java's DNS based package naming convention and XML's URI based namespaces.

YAML was designed to support incremental interfaces that includes both input pull-style and output push-style one-pass (SAX-like) interfaces. Together these enable YAML to support the processing of large documents, such as a transaction log, or continuous streams, such as a feed from a production machine.

1.3. Relation to XML

Newcomers to YAML often search for its correlation to the eXtensible Markup Language (XML). While the two languages may actually compete in several application domains, there is no direct correlation between them.

YAML is primarily a data serialization language. XML was designed to be backwards compatible with the Standard Generalized Markup Language (SGML) and thus had many design constraints placed on it that YAML does not share. Inheriting SGML's legacy, XML is designed to support structured documentation, where YAML is more closely targeted at data structures and messaging. Where XML is a pioneer in many domains, YAML is the result of lessons learned from XML and other technologies.

It should be mentioned that there are ongoing efforts to define standard XML/YAML mappings. This generally requires that a subset of each language be used. For more information on using both XML and YAML, please visit http://yaml.org/xml/index.html.

1.4. Terminology

This specification uses key words based on RFC2119 [http://www.ietf.org/rfc/rfc2119.txt] to indicate requirement level. In particular, the following words are used to describe the actions of a YAML processor:

May	The word *may*, or the adjective *optional*, mean that conforming YAML processors are permitted, but *need not* behave as described.
Should	The word *should*, or the adjective *recommended*, mean that there could be reasons for a YAML processor to deviate from the behavior described, but that such deviation could hurt interoperability and should therefore be advertised with appropriate notice.
Must	The word *must*, or the term *required* or *shall*, mean that the behavior described is an absolute requirement of the specification.

Chapter 2. Preview

This section provides a quick glimpse into the expressive power of YAML. It is not expected that the first-time reader grok all of the examples. Rather, these selections are used as motivation for the remainder of the specification.

2.1. Collections

YAML's block collections use indentation for scope and begin each entry on its own line. Block sequences indicate each entry with a dash and space (“`-`”). Mappings use a colon and space (“`: `”) to mark each mapping key: value pair.

Example 2.1. Sequence of Scalars (ball players)

```
- Mark McGwire
- Sammy Sosa
- Ken Griffey
```

Example 2.2. Mapping Scalars to Scalars (player statistics)

```
hr:  65
avg: 0.278
rbi: 147
```

Example 2.3. Mapping Scalars to Sequences (ball clubs in each league)

```
american:
  - Boston Red Sox
  - Detroit Tigers
  - New York Yankees
national:
  - New York Mets
  - Chicago Cubs
  - Atlanta Braves
```

Example 2.4. Sequence of Mappings (players' statistics)

```
-
  name: Mark McGwire
  hr:   65
  avg:  0.278
-
  name: Sammy Sosa
  hr:   63
  avg:  0.288
```

YAML also has flow styles, using explicit indicators rather than indentation to denote scope. The flow sequence is written as a comma separated list within square brackets. In a similar manner, the flow mapping uses curly braces.

Example 2.5. Sequence of Sequences

```
- [name        , hr, avg  ]
- [Mark McGwire, 65, 0.278]
- [Sammy Sosa  , 63, 0.288]
```

Example 2.6. Mapping of Mappings

```
Mark McGwire: {hr: 65, avg: 0.278}
Sammy Sosa: {
    hr: 63,
    avg: 0.288
  }
```

2.2. Structures

YAML uses three dashes (“`---`”) to separate documents within a stream. Three dots (“`...`”) indicate the end of a document without starting a new one, for use in communication channels. Comment lines begin with the Octothorpe (usually called the “hash” or “pound” sign - “`#`”).

Example 2.7. Two Documents in a Stream (each with a leading comment)

```
# Ranking of 1998 home runs
---
- Mark McGwire
- Sammy Sosa
- Ken Griffey

# Team ranking
---
- Chicago Cubs
- St Louis Cardinals
```

Example 2.8. Play by Play Feed from a Game

```
---
time: 20:03:20
player: Sammy Sosa
action: strike (miss)
...
---
time: 20:03:47
player: Sammy Sosa
action: grand slam
...
```

Repeated nodes are first identified by an anchor (marked with the ampersand - “`&`”), and are then aliased (referenced with an asterisk - “`*`”) thereafter.

Example 2.9. Single Document with Two Comments

```
---
hr: # 1998 hr ranking
  - Mark McGwire
  - Sammy Sosa
rbi:
  # 1998 rbi ranking
  - Sammy Sosa
  - Ken Griffey
```

Example 2.10. Node for “`Sammy Sosa`” appears twice in this document

```
---
hr:
  - Mark McGwire
  # Following node labeled SS
  - &SS Sammy Sosa
rbi:
  - *SS # Subsequent occurrence
  - Ken Griffey
```

A question mark and space (“**?** ”) indicate a complex mapping key. Within a block collection, key: value pairs can start immediately following the dash, colon or question mark.

Example 2.11. Mapping between Sequences

```
? - Detroit Tigers
  - Chicago cubs
:
  - 2001-07-23

? [ New York Yankees,
    Atlanta Braves ]
: [ 2001-07-02, 2001-08-12,
    2001-08-14 ]
```

Example 2.12. In-Line Nested Mapping

```
---
# products purchased
- item    : Super Hoop
  quantity: 1
- item    : Basketball
  quantity: 4
- item    : Big Shoes
  quantity: 1
```

2.3. Scalars

Scalar content can be written in block form using a literal style (“**|**”) where all line breaks count. Or they can be written with the folded style (“**>**”) where each line break is folded to a space unless it ends an empty or a “more indented” line.

Example 2.13. In literals, newlines are preserved

```
# ASCII Art
--- |
  \//||\/||
  // ||  ||__
```

Example 2.14. In the plain scalar, newlines become spaces

```
---
  Mark McGwire's
  year was crippled
  by a knee injury.
```

Example 2.15. Folded newlines preserved for "more indented" and blank lines

```
>
 Sammy Sosa completed another
 fine season with great stats.

   63 Home Runs
   0.288 Batting Average

 What a year!
```

Example 2.16. Indentation determines scope

```
name: Mark McGwire
accomplishment: >
  Mark set a major league
  home run record in 1998.
stats: |
  65 Home Runs
  0.278 Batting Average
```

YAML's flow scalars include the plain style (most examples thus far) and quoted styles. The double quoted style provides escape sequences. The single quoted style is useful when escaping is not needed. All flow scalars can span multiple lines; line breaks are always folded.

Example 2.17. Quoted Scalars

```
unicode: "Sosa did fine.\u263A"
control: "\b1998\t1999\t2000\n"
hexesc:  "\x13\x10 is \r\n"

single: '"Howdy!" he cried.'
quoted: ' # not a ''comment''.'
tie-fighter: '|\-*-/|'
```

Example 2.18. Multi-line Flow Scalars

```
plain:
  This unquoted scalar
  spans many lines.

quoted: "So does this
  quoted scalar.\n"
```

2.4. Tags

In YAML, untagged nodes are given an type depending on the application. The examples in this specification generally use the “**`seq`**” [http://yaml.org/type/seq.html], “**`map`**” [http://yaml.org/type/map.html] and “**`str`**” [http://yaml.org/type/str.html] types from the YAML tag repository [http://yaml.org/type/index.html]. A few examples also use the “**`int`**” [http://yaml.org/type/int.html] and “**`float`**” [http://yaml.org/type/float.html] types. The repository includes additional types such as “**`null`**” [http://yaml.org/type/null.html], “**`bool`**” [http://yaml.org/type/bool.html], “**`set`**” [http://yaml.org/type/set.html] and others.

Example 2.19. Integers

```
canonical: 12345
decimal: +12,345
sexagecimal: 3:25:45
octal: 014
hexadecimal: 0xC
```

Example 2.20. Floating Point

```
canonical: 1.23015e+3
exponential: 12.3015e+02
sexagecimal: 20:30.15
fixed: 1,230.15
negative infinity: -.inf
not a number: .NaN
```

Example 2.21. Miscellaneous

```
null: ~
true: y
false: n
string: '12345'
```

Example 2.22. Timestamps

```
canonical: 2001-12-15T02:59:43.1Z
iso8601: 2001-12-14t21:59:43.10-05:00
spaced: 2001-12-14 21:59:43.10 -5
date: 2002-12-14
```

Explicit typing is denoted with a tag using the exclamantion point ("**!**") symbol. Global tags are URIs and may be specified in a shorthand form using a handle. Application-specific local tags may also be used.

Example 2.23. Various Explicit Tags

```
---
not-date: !!str 2002-04-28

picture: !!binary |
 R0lGODlhDAAMAIQAAP//9/X
 17unp5WZmZgAAAOfn515eXv
 Pz7Y6OjuDg4J+fn5OTk6enp
 56enmleECcgggoBADs=

application specific tag: !something |
 The semantics of the tag
 above may be different for
 different documents.
```

Example 2.24. Global Tags

```
%TAG ! tag:clarkevans.com,2002:
--- !shape
  # Use the ! handle for presenting
  # tag:clarkevans.com,2002:circle
- !circle
  center: &ORIGIN {x: 73, y: 129}
  radius: 7
- !line
  start: *ORIGIN
  finish: { x: 89, y: 102 }
- !label
  start: *ORIGIN
  color: 0xFFEEBB
  text: Pretty vector drawing.
```

Example 2.25. Unordered Sets

```
# sets are represented as a
# mapping where each key is
# associated with the empty string
--- !!set
? Mark McGwire
? Sammy Sosa
? Ken Griff
```

Example 2.26. Ordered Mappings

```
# ordered maps are represented as
# a sequence of mappings, with
# each mapping having one key
--- !!omap
- Mark McGwire: 65
- Sammy Sosa: 63
- Ken Griffy: 58
```

2.5. Full Length Example

Below are two full-length examples of YAML. On the left is a sample invoice; on the right is a sample log file.

Example 2.27. Invoice

```
--- !<tag:clarkevans.com,2002:invoice>
invoice: 34843
date    : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
        state   : MI
        postal  : 48046
ship-to: *id001
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
total: 4443.52
comments:
    Late afternoon is best.
    Backup contact is Nancy
    Billsmer @ 338-4338.
```

Example 2.28. Log File

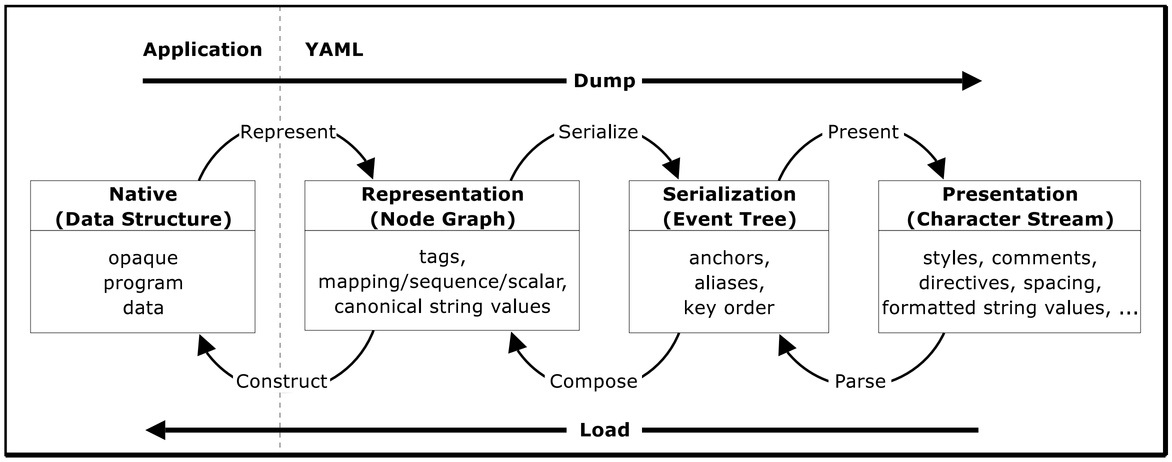
```
---
Time: 2001-11-23 15:01:42 -5
User: ed
Warning:
  This is an error message
  for the log file
---
Time: 2001-11-23 15:02:31 -5
User: ed
Warning:
  A slightly different error
  message.
---
Date: 2001-11-23 15:03:17 -5
User: ed
Fatal:
  Unknown variable "bar"
Stack:
  - file: TopClass.py
    line: 23
    code: |
      x = MoreObject("345\n")
  - file: MoreClass.py
    line: 58
    code: |-
      foo = bar
```

Chapter 3. Processing YAML Information

YAML is both a text format and a method for presenting any data structure in this format. Therefore, this specification defines two concepts: a class of data objects called YAML representations, and a syntax for presenting YAML representations as a series of characters, called a YAML stream. A YAML *processor* is a tool for converting information between these complementary views. It is assumed that a YAML processor does its work on behalf of another module, called an *application*. This chapter describes the information structures a YAML processor must provide to or obtain from the application.

YAML information is used in two ways: for machine processing, and for human consumption. The challenge of reconciling these two perspectives is best done in three distinct translation stages: representation, serialization, and presentation. Representation addresses how YAML views native data structures to achieve portability between programming environments. Serialization concerns itself with turning a YAML representation into a serial form, that is, a form with sequential access constraints. Presentation deals with the formatting of a YAML serialization as a series of characters in a human-friendly manner.

Figure 3.1. Processing Overview


Application
YAML
Dump
Represent
Serialize
Present
Native (Data Structure)
opaque program data
Representation (Node Graph)
tags, mapping/sequence/scalar, canonical string values
Serialization (Event Tree)
anchors, aliases, key order
Presentation (Character Stream)
styles, comments, directives, spacing, formatted string values, ...
Construct
Compose
Parse
Load

A YAML processor need not expose the serialization or representation stages. It may translate directly between native data structures and a character stream (*dump* and *load* in the diagram above). However, such a direct translation should take place so that the native data structures are constructed only from information available in the representation.

3.1. Processes

This section details the processes shown in the diagram above. Note a YAML processor need not provide all these processes. For example, a YAML library may provide only YAML input ability, for loading configuration files, or only output ability, for sending data to other applications.

3.1.1. Represent

YAML *represents* any native data structure using three node kinds: the sequence, the mapping and the scalar. By sequence we mean an ordered series of entries, by mapping we mean an unordered association of unique keys to values, and by scalar we mean any datum with opaque structure presentable as a series of Unicode characters. Combined, these primitives generate directed graph structures. These primitives were chosen because they are both powerful and familiar: the sequence corresponds to a Perl array and a Python list, the mapping corresponds to a Perl hash table and a Python dictionary. The scalar represents strings, integers, dates and other atomic data types.

Each YAML node requires, in addition to its kind and content, a tag specifying its data type. Type specifiers are either global URIs, or are local in scope to a single application. For example, an integer is represented in YAML with a scalar plus the global tag "**`tag:yaml.org,2002:int`**". Similarly, an invoice object, particular to a given organization, could be represented as a mapping together with the local tag "**`!invoice`**". This simple model can represent any data structure independent of programming language.

3.1.2. Serialize

For sequential access mediums, such as an event callback API, a YAML representation must be *serialized* to an ordered tree. Since in a YAML representation, mapping keys are unordered and nodes may be referenced more than once (have more than one incoming "arrow"), the serialization process is required to impose an ordering on the mapping keys and to replace the second and subsequent references to a given node with place holders called aliases. YAML does not specify how these *serialization details* are chosen. It is up to the YAML processor to come up with human-friendly key order and anchor names, possibly with the help of the application. The result of this process, a YAML serialization tree, can then be traversed to produce a series of event calls for one-pass processing of YAML data.

3.1.3. Present

The final output process is *presenting* the YAML serializations as a character stream in a human-friendly manner. To maximize human readability, YAML offsers a rich set of stylistic options which go far beyond the minimal functional needs of simple data storage. Therefore the YAML processor is required to introduce various *presentation details* when creating the stream, such as the choice of node styles, how to format content, the amount of indentation, which tag handles to use, the node tags to leave unspecified, the set of directives to provide and possibly even what comments to add. While some of this can be done with the help of of the application, in general this process should guided by the preferences of the user.

3.1.4. Parse

Parsing is the inverse process of presentation, it takes a stream of characters and produces a series of events. Parsing discards all the details introduced in the presentation process, reporting only the serialization events. Parsing can fail fue to ill-formed input.

3.1.5. Compose

Composing takes a series of serialization events and produces a representation graph. Composing discards all the serialization details introduced in the serialization process, producing only the representation graph. Composing can fail due to any of several reasons, detailed below.

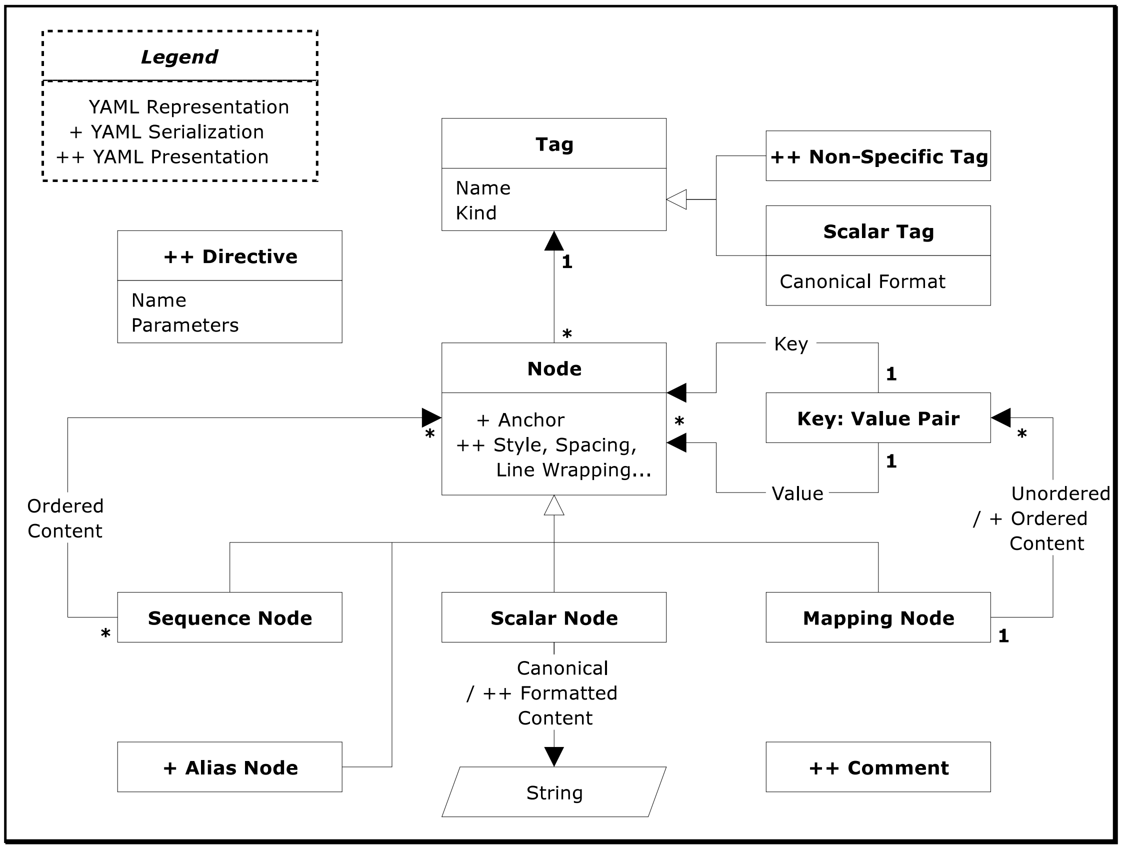
3.1.6. Construct

The final input process is *constructing* native data structures from the YAML representation. Construction must be based only on the information available in the representation, and not on additional serialization or presentation details such as comments, directives, mapping key order, node styles, content format, indentation levels etc. Construction can fail due to the unavailability of the required native data types.

3.2. Information Models

This section specifies the formal details of the results of the above processes. To maximize data portability between programming languages and implementations, users of YAML should be mindful of the distinction between serialization or presentation properties and those which are part of the YAML representation. Thus, while imposing a order on mapping keys is necessary for flattening YAML representations to a sequential access medium, this serialization detail must not be used to convey application level information. In a similar manner, while indentation technique and a choice of a node style are needed for the human readability, these presentation details are neither part of the YAML serialization nor the YAML representation. By carefully separating properties needed for serialization and presentation, YAML representations of application information will be consistent and portable between various programming environments.

Figure 3.2. Information Models

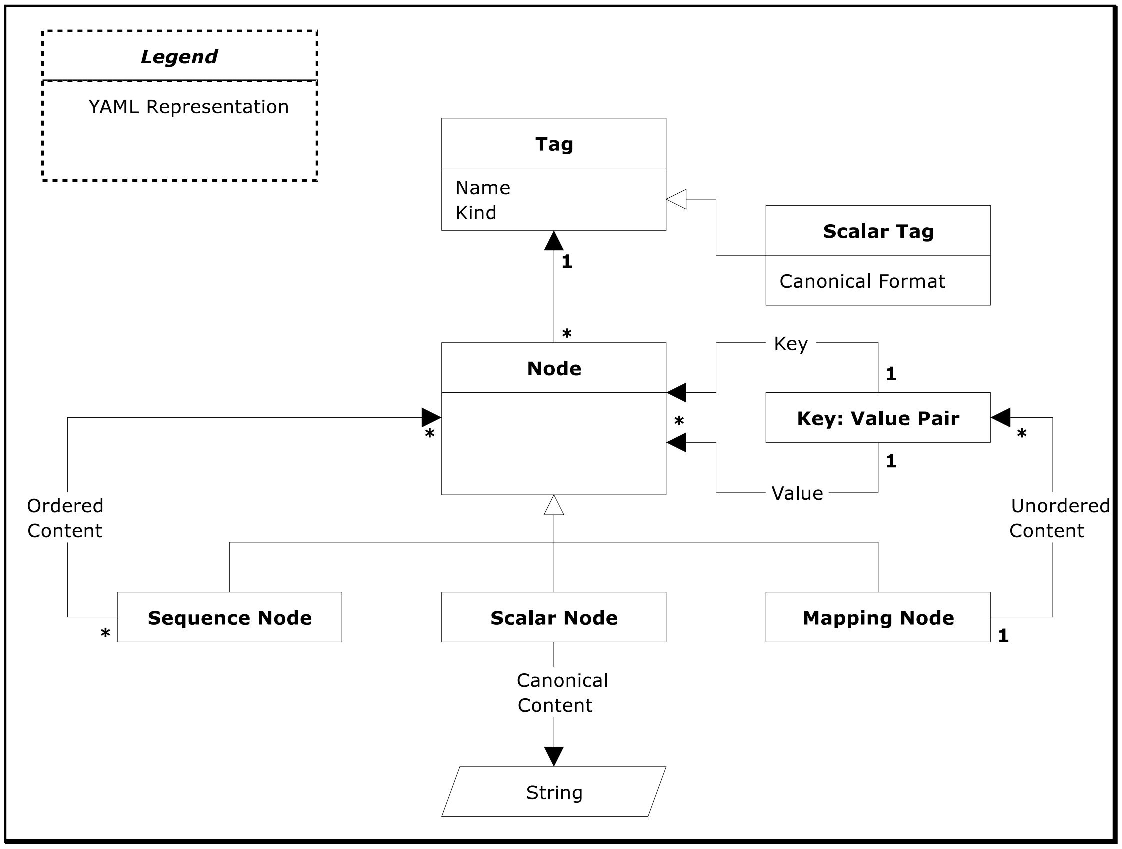


3.2.1. Representation Graph

YAML's *representation* of native data is a rooted, connected, directed graph of tagged nodes. By “directed graph” we mean a set of nodes and directed edges (“arrows”), where each edge connects one node to another (see a formal definition [http://www.nist.gov/dads/HTML/directedGraph.html]). All the nodes must be reachable from the *root node* via such edges. Note that the YAML graph may include cycles, and a node may have more than one incoming edge.

Nodes that are defined in terms of other nodes are collections and nodes that are independent of any other nodes are scalars. YAML supports two kinds of collection nodes, sequences and mappings. Mapping nodes are somewhat tricky because their keys are unordered and must be unique.

Figure 3.3. Representation Model



3.2.1.1. Nodes

YAML *nodes* have *content* of one of three *kinds*: scalar, sequence, or mapping. In addition, each node has a tag which serves to restrict the set of possible values which the node's content can have.

Scalar	The content of a *scalar* node is an opaque datum that can be presented as a series of zero or more Unicode characters.
Sequence	The content of a *sequence* node is an ordered series of zero or more nodes. In particular, a sequence may contain the same node more than once or it could even contain itself (directly or indirectly).
Mapping	The content of a *mapping* node is an unordered set of *key: value* node pairs, with the restriction that each of the keys is unique. YAML places no further restrictions on the nodes. In particular, keys may be arbitrary nodes, the same node may be used as the value of several key: value pairs, and a mapping could even contain itself as a key or a value (directly or indirectly).

When appropriate, it is convenient to consider sequences and mappings together, as *collections*. In this view, sequences are treated as mappings with integer keys starting at zero. Having a unified collections view for sequences and mappings is helpful both for creating practical YAML tools and APIs and for theoretical analysis.

3.2.1.2. Tags

YAML represents type information of native data structures with a simple identifier, called a *tag*. *Global tags* are are URIs [http://www.ietf.org/rfc/rfc2396.txt] and hence globally unique across all applications. The “**`tag`**”: URI scheme [http://www.taguri.org] (mirror [http://yaml.org/spec/taguri.txt]) is recommended for all global YAML tags. In contrast, *local tags* are specific to a single application. Local tags start with "`!`", are not URIs and are not expected to be globally unique. YAML provides a "**`TAG`**" directive to make tag notation less verbose; it also offers easy migration from local to global tags. To ensure this, local tags are restricted to the URI character set and use URI character escaping.

YAML does not mandate any special relationship between different tags that begin with the same substring. Tags ending with URI fragments (containing "`#`") are no exception; tags that share the same base URI but differ in their fragment part are considered to be different, independent tags. By convention, fragments are used to identify different "variants" of a tag, while "`/`" is used to define nested tag "namespace" hierarchies. However, this is merely a convention, and each tag may employ its own rules. For example, Perl tags may use "`::`" to express namespace hierarchies, Java tags may use "`.`", etc.

YAML tags are used to associate meta information with each node. In particular, each tag must specify the expected node kind (scalar, sequence, or mapping). Scalar tags must also provide mechanism for converting formatted content to a canonical form for supporting equality testing. Furthermore, a tag may provide additional information such as the set of allowed content values for validation, a mechanism for tag resolution, or any other data that is applicable to all of the tag's nodes.

3.2.1.3. Nodes Comparison

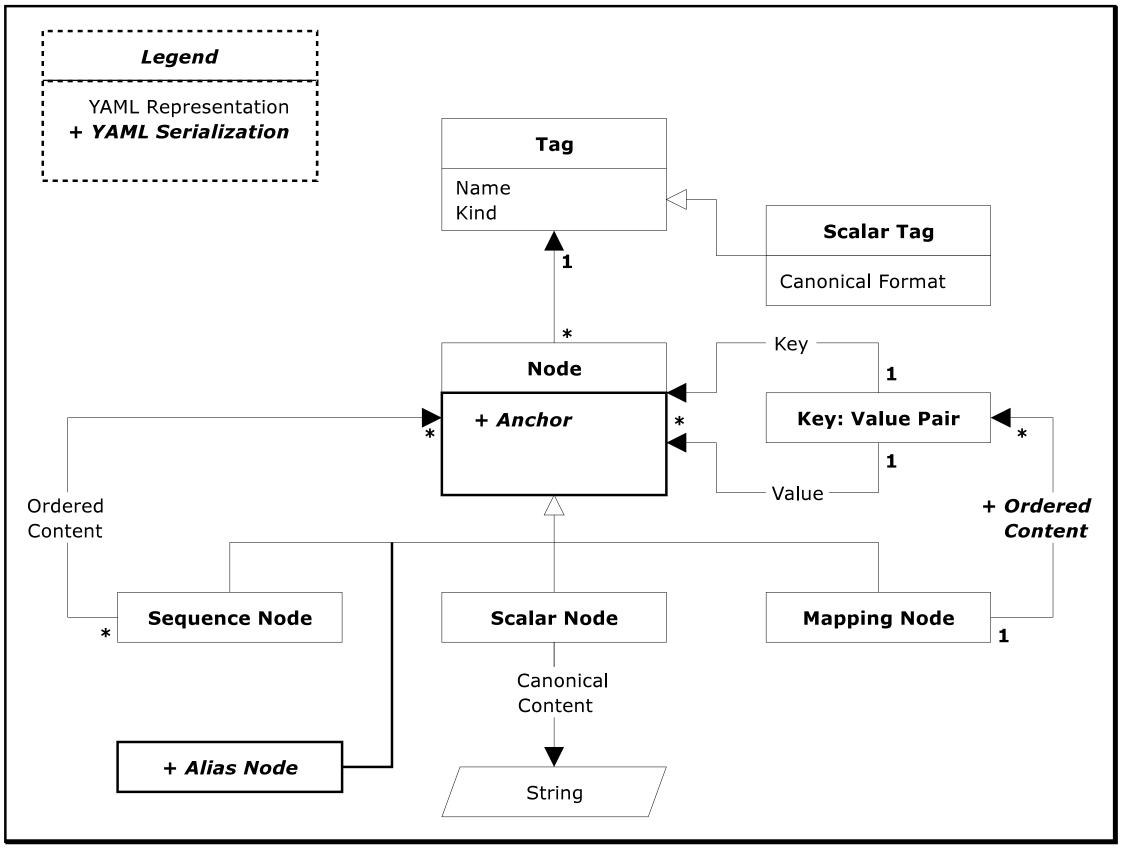
Since YAML mappings require key uniqueness, representations must include a mechanism for testing the equality of nodes. This is non-trivial since YAML allows various ways to format a given scalar content. For example, the integer eleven can be written as "**`013`**" (octal) or "**`0xB`**" (hexadecimal). If both forms are used as keys in the same mapping, only a YAML processor which recognizes integer formats would correctly flag the duplicate key as an error.

Canonical Form	YAML supports the need for scalar equality by requiring that every scalar tag must specify a mechanism to producing the *canonical form* of any formatted content. This form is a Unicode character string which presents the content and can be used for equality testing. While this requirement is stronger than a well defined equality operator, it has other uses, such as the production of digital signatures.
Equality	Two nodes must have the same tag and content to be *equal*. Since each tag applies to exactly one kind, this implies that the two nodes must have the same kind to be equal. Two scalars are equal only when their tags and canonical forms are equal character-by-character. Equality of collections is defined recursively. Two sequences are equal only when they have the same tag and length, and each node in one sequence is equal to the corresponding node in the other sequence. Two mappings are equal only when they have the same tag and an equal set of keys, and each key in this set is associated with equal values in both mappings.
Identity	Two nodes are *identical* only when they represent the same native data structure. Typically, this corresponds to a single memory address. Identity should not be confused with equality; two equal nodes need not have the same identity. A YAML processor may treat equal scalars as if they were identical. In contrast, the separate identity of two distinct but equal collections must be preserved.

3.2.2. Serialization Tree

To express a YAML representation using a serial API, it necessary to impose an order on mapping keys and employ alias nodes to indicate a subsequent occurrence of a previously encountered node. The result of this process is a *serialization tree*, where each node has an ordered set of children. This tree can be traversed for a serial event based API. Construction of native structures from the serial interface should not use key order or anchors for the preservation of important data.

Figure 3.4. Serialization Model



3.2.2.1. Keys Order

In the representation model, mapping keys do not have an order. To serialize a mapping, it is necessary to impose an *ordering* on its keys. This order is a serialization detail and should not be used when composing the representation graph (and hence for the preservation of important data). In every case where node order is significant, a sequence must be used. For example, an ordered mapping can be represented as a sequence of mappings, where each mapping is a single key: value pair. YAML provides convenient compact notation for this case.

3.2.2.2. Anchors and Aliases

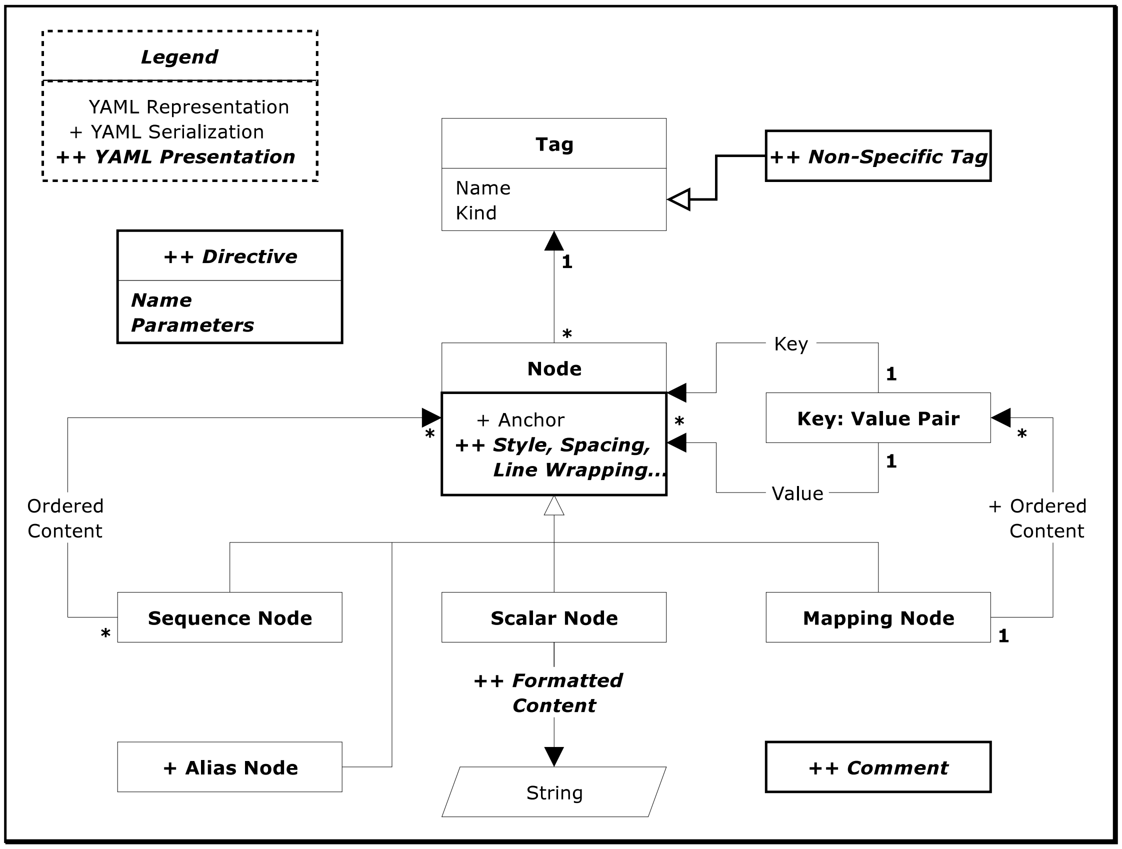
In the representation graph, a node may appear in more than one collection. When serializing such data, the first occurrence of the node is *identified* by an *anchor* and each subsequent occurrence is serialized as an *alias node* which refers back to this anchor. Otherwise, anchor names are a serialization detail and are discarded once composing is completed. When composing a representation graph from serialized events, an alias node refers to the most recent node in the serialization having the specified anchor. Therefore, anchors need not be unique within a serialization. In addition, an anchor need not have an alias node referring to it. It is therefore possible to provide an anchor for all nodes in serialization.

3.2.3. Presentation Stream

A YAML *presentation* is a *stream* of Unicode characters making use of of styles, formats, comments, directives and other presentation details to present a YAML serialization in a human readable way. Although a YAML processor may provide these details when parsing, they should not be reflected in the resulting serialization. YAML allows several serializations to be contained in the same YAML character stream as a series of *documents* separated by document boundary markers.

Documents appearing in the same stream are independent; that is, a node must not appear in more than one serialization tree or representation graph.

Figure 3.5. Presentation Model



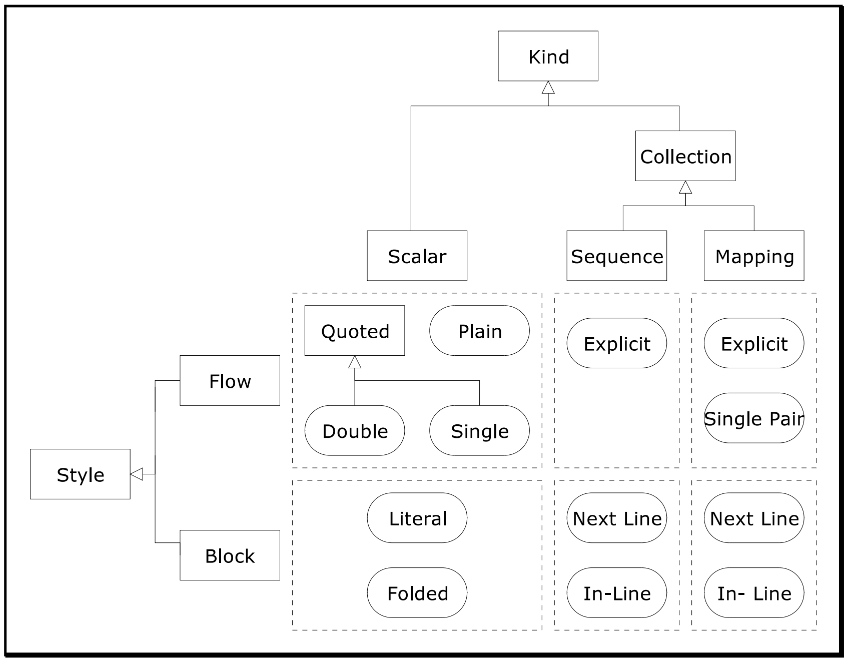
3.2.3.1. Node Styles

Each node is presented in some *style*, depending on its kind. The node style is a presentation detail and is not reflected in the serialization tree or representation graph. There are two groups of styles, *block* and *flow*. Block styles use indentation to denote nesting and scope within the document. In contrast, flow styles rely on explicit indicators to denote nesting and scope.

YAML provides a rich set of scalar styles. *Block scalar styles* include the *literal style* and the *folded style*; *flow scalar styles* include the *plain style* and two *quoted styles*, the *single quoted style* and the *double quoted style*. These styles offer a range of trade-offs between expressive power and readability.

Normally, the content of *block collections* begins on the next line. In most cases, YAML also allows block collections to start *in-line* for more compact notation when nesting *block sequences* and *block mappings* inside each other. When nesting *flow collections*, a *flow mapping* with a *single key: value pair* may be specified directly inside a *flow sequence*, allowing for a compact “ordered mapping” notation.

Figure 3.6. Kind/Style Combinations



3.2.3.2. Scalar Formats

YAML allows scalar content to be presented in several *formats*. For example, the boolean “**`true`**” might also be written as “**`yes`**”. Tags must specify a mechanism for converting any formatted scalar content to a canonical form for use in equality testing. Like node style, the format is a presentation detail and is not reflected in the serialization tree and representation graph.

3.2.3.3. Comments

Comments are a presentation detail and must not have any effect on the serialization tree or representation graph. In particular, comments are not associated with a particular node. The usual purpose of a comment is to communicate between the human maintainers of a file. A typical example is comments in a configuration file. Comments may not appear inside scalars, but may be interleaved with such scalars inside collections.

3.2.3.4. Directives

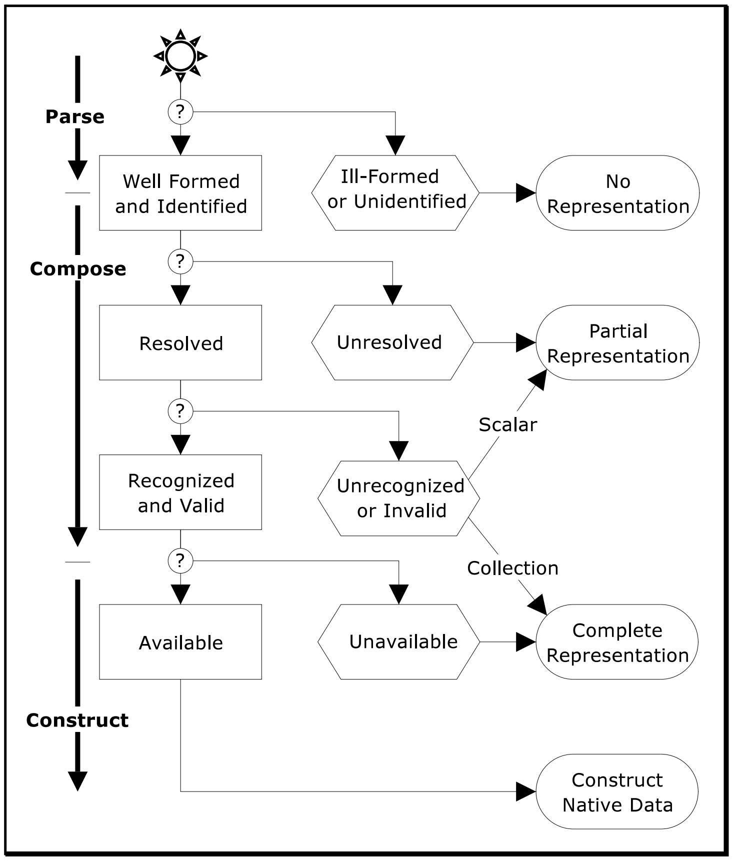
Each document may be associated with a set of *directives*. A directive has a name and an optional sequence of parameters. Directives are instructions to the YAML processor, and like all other presentation details are not reflected in the YAML serialization tree or representation graph. This version of YAML defines a two directives, “**`YAML`**” and “**`TAG`**”. All other directives are reserved for future versions of YAML.

3.3. Loading Failure Points

The process of loading native data structures from a YAML stream has several potential *failure points*. The character stream may be ill-formed, aliases may be unidentified, unspecified tags may be unresolvable, tags may be unrecognized, the content may be invalid, and a native type may be unavailable. Each of these failures results with an incomplete loading.

A *partial representation* need not resolve the tag of each node, and the canonical form of scalar content need not be available. This weaker representation is useful for cases of incomplete knowledge of the types used in the document. In contrast, a *complete representation* specifies the tag of each node, and provides the canonical form of scalar content, allowing for equality testing. A complete representation is required in order to construct native data structures.

Figure 3.7. Loading Failure Points



3.3.1. Well-Formed and Identified

A *well-formed* character stream must match the productions specified in the next chapter. Successful loading also requires that each alias shall refer to a previous node identified by the anchor. A YAML processor should reject *ill-formed streams* and *unidentified aliases*. A YAML processor may recover from syntax errors, possibly by ignoring certain parts of the input, but it must provide a mechanism for reporting such errors.

3.3.2. Resolved

It is not required that all the tags of the complete representation be explicitly specified in the character stream. During parsing, nodes that omit the tag are given a *non-specific tag*: “**?**” for plain scalars and “**!**” for all other nodes. These non-specific tags must be *resolved* to a *specific tag* (either a local tag or a global tag) for a complete representation to be composed.

Resolving the tag of a node must only depend on the following three parameters: the non-specific tag of the node, the path leading from the root node to the node, and the content (and hence the kind) of the node. In particular, resolution must not consider presentation details such as comments, indentation and node style. Also, resolution must not consider the content of any other node, except for the content of the key nodes directly along the path leading from the root node to the resolved node. In particular, resolution must not consider the content of a sibling node in a collection or the content of the value node associated with a resolved key node.

Tag resolution is specific to the application, hence a YAML processor should provide a mechanism allowing the application to specify the tag resolution rules. It is recommended that nodes having the “**`!`**” non-specific tag should be resolved as “**`tag:yaml.org,2002:seq`**”, “**`tag:yaml.org,2002:map`**” or “**`tag:yaml.org,2002:str`**” depending on the node's kind. This convention allows the author of a YAML character stream to exert some measure of control over the tag resolution process. By explicitly specifying a plain scalar has the “**`!`**” non-specific tag, the node is resolved as a string, as if it was quoted or written in a block style. Note, however, that each application may override this behavior. For example, an application may automatically detect the type of programming language used in source code presented as a non-plain scalar and resolve it accordingly.

When a node has more than one occurence (using an anchor and alias nodes), tag resolution must depend only on the path to the first occurence of the node. Typically, the path leading to a node is sufficient to determine its specific tag. In cases where the path does not imply a single specific tag, the resolution also needs to consider the node content to select amongst the set of possible tags. Thus, plain scalars may be matched against a set of regular expressions to provide automatic resolution of integers, floats, timestamps and similar types. Similarly, the content of mapping nodes may be matched against sets of expected keys to automatically resolve points, complex numbers and similar types.

The combined effect of these rules is to ensure that tag resolution can be performed as soon as a node is first encountered in the stream, typically before its content is parsed. Also, tag resolution only requires refering to a relatively small number of previously parsed nodes. Thus, tag resolution in one-pass processors is both possible and practical.

If a document contains *unresolved tags*, the YAML processor is unable to compose a complete representation graph. In such a case, the YAML processor may compose an partial representation, based on each node's kind and allowing for non-specific tags.

3.3.3. Recognized and Valid

To be *valid*, a node must have a tag which is *recognized* by the YAML processor and its content must satisfy the constraints imposed by this tag. If a document contains a scalar node with an *unrecognized tag* or *invalid content*, only a partial representation may be composed. In contrast, a YAML processor can always compose a complete representation for an unrecognized or an invalid collection, since collection equality does not depend upon knowledge of the collection's data type. However, such a complete representation can not be used to construct a native data structure.

3.3.4. Available

In a given processing environment, there need not be an *available* native type corresponding to a given tag. If a node's tag is *unavailable*, a YAML processor will not be able to construct a native data structure for it. In this case, a complete representation may still be composed, and an application may wish to use this representation directly.

Chapter 4. Syntax

Following are the BNF productions defining the syntax of YAML character streams. To make this chapter easier to follow, production names use Hungarian-style notation:

`e-`	A production matching no characters.
`c-`	A production matching one or more characters starting and ending with a special (non-space) character.
`b-`	A production matching a single line break.
`nb-`	A production matching one or more characters starting and ending with a non-break character.
`s-`	A production matching one or more characters starting and ending with a space character.
`ns-`	A production matching one or more characters starting and ending with a non-space character.
`X-Y-`	A production matching a sequence of one or more characters, starting with an `X-` character and ending with a `Y-` character.
`l-`	A production matching one or more lines (shorthand for **`s-b-`**).
`X+`, `X-Y+`	A production as above, with the additional property that the indentation level used is greater than the specified `n` parameter.

Productions are generally introduced in a “bottom-up” order; basic productions are specified before the more complex productions using them. Examples accompanying the productions list display sample YAML text side-by-side with equivalent YAML text using only flow collections and double quoted scalars. For improved readability, the equivalent YAML text uses the “**`!!seq`**”, “**`!!map`**” and “**`!!str`**” shorthands instead of the verbatim “**`!<tag:yaml.org,2002:seq>`**”, “**`!<tag:yaml.org,2002:map>`**” and “**`!<tag:yaml.org,2002:str>`**” forms. These types are used to resolve all untagged nodes, except for a few examples that use the “**`!!int`**” and “**`!!float`**” types.

4.1. Characters

4.1.1. Character Set

YAML streams use the *printable* subset of the Unicode character set. On input, a YAML processor must accept all printable ASCII characters, the space, tab, line break, and all Unicode characters beyond #x9F. On output, a YAML processor must only produce these acceptable characters, and should also escape all non-printable Unicode characters. The allowed character range explicitly excludes the surrogate block **`#xD800-#xDFFF`**, DEL **`#x7F`**, the C0 control block **`#x0-#x1F`**, the C1 control block **`#x80-#x9F`**, **`#xFFFE`** and **`#xFFFF`**. Any such characters must be presented using escape sequences.

```
[1]  c-printable ::=   #x9 | #xA | #xD | [#x20-#x7E]          /* 8 bit */
                     | #x85 | [#xA0-#xD7FF] | [#xE000-#xFFFD] /* 16 bit */
                     | [#x10000-#x10FFFF]                     /* 32 bit */
```

4.1.2. Character Encoding

All characters mentioned in this specification are Unicode code points. Each such code point is written as one or more octets depending on the *character encoding* used. Note that in UTF-16, characters above **`#xFFFF`** are written as four octets, using a surrogate pair. A YAML processor must support the UTF-16 and UTF-8 character encodings. If a character stream does not begin with a *byte order mark* (**`#FEFF`**), the character encoding shall be UTF-8. Otherwise it shall be either UTF-8, UTF-16 LE or UTF-16 BE as indicated by the byte order mark. On output, it is recommended that a byte order mark should only be emitted for UTF-16 character encodings. Note that the UTF-32 encoding is explicitly not supported. For more information about the byte order mark and the Unicode character encoding schemes see the Unicode FAQ [http://www.unicode.org/unicode/faq/utf_bom.html].

[2] `c-byte-order-mark ::= #xFEFF`

In the examples, byte order mark characters are displayed as “⇔”.

Example 4.1. Byte Order Mark

```
⇔# Comment only.
```

```
# This stream contains no
# documents, only comments.
```

```
Legend:
  c-byte-order-mark
```

Example 4.2. Invalid Byte Order Mark

```
# Invalid use of BOM
⇔# inside a
# document.
```

```
ERROR:
 A BOM must not appear
 inside a document.
```

4.1.3. Indicator Characters

Indicators are characters that have special semantics used to describe the structure and content of a YAML document.

- A “**-**” denotes a blocks equence entry.

[3] `c-sequence-entry ::= “-”`

- A “**?**” denotes a mapping key.

[4] `c-mapping-key ::= “?”`

- A “**:**” denotes a mapping value.

[5] `c-mapping-value ::= “:”`

Example 4.3. Block Structure Indicators

```
sequence:
- one
- two
mapping:
  ? sky
  : blue
  ? sea : green
```

```
Legend:
  c-sequence-entry
  c-mapping-key
  c-mapping-value
```

```
%YAML 1.1
---
!!map {
  ? !!str "sequence"
  : !!seq [
    !!str "one", !!str "two"
  ],
  ? !!str "mapping"
  : !!map {
    ? !!str "sky" : !!str "blue",
    ? !!str "sea" : !!str "green",
  }
}
```

- A “**,**” ends a flow collection entry.

[6] `c-collect-entry ::= “,”`

- A “**[**” starts a flow sequence.

[7] `c-sequence-start ::= “[”`

- A “**]**” ends a flow sequence.

[8] `c-sequence-end ::= “]”`

- A “**{**” starts a flow mapping.

[9] `c-mapping-start ::= “{”`

- A “**}**” ends a flow mapping.

[10] `c-mapping-end ::= “}”`

Example 4.4. Flow Collection Indicators

```
sequence: [ one, two, ]
mapping: { sky: blue, sea: green }
```

```
Legend:
  c-sequence-start  c-sequence-end
  c-mapping-start   c-mapping-end
  c-collect-entry
```

```
%YAML 1.1
---
!!map {
  ? !!str "sequence"
  : !!seq [
    !!str "one", !!str "two"
  ],
  ? !!str "mapping"
  : !!map {
    ? !!str "sky" : !!str "blue",
    ? !!str "sea" : !!str "green",
  }
}
```

- A “ **#**” denotes a comment.

[11] `c-comment ::= “#”`

Example 4.5. Comment Indicator

```
# Comment only.
```

```
Legend:
  c-comment
```

```
# This stream contains no
# documents, only comments.
```

- A “**&**” denotes a node's anchor property.

[12] `c-anchor ::= “&”`

- A “*****” denotes an alias node.

[13] `c-alias ::= “*”`

- A “**!**” denotes a node's tag.

[14] `c-tag ::= “!”`

Example 4.6. Node Property Indicators

```
anchored: !local &anchor value
alias: *anchor
```

```
Legend:
  c-anchor
  c-alias
  c-tag
```

```
%YAML 1.1
---
!!map {
  ? !!str "anchored"
  : !local &A1 "value",
  ? !!str "alias"
  : *A1,
}
```

- A “**|**” denotes a literal block scalar.

[15] `c-literal ::= “|”`

- A “**>**” denotes a folded block scalar.

[16] `c-folded ::= “>”`

Example 4.7. Block Scalar Indicators

```
literal: |
  text
folded: >
  text
```

```
Legend:
  c-literal
  c-folded
```

```
%YAML 1.1
---
!!map {
  ? !!str "literal"
  : !!str "text\n",
  ? !!str "folded"
  : !!str "text\n",
}
```

- A “**'**” surrounds a single quoted flow scalar.

[17] `c-single-quote ::= “'”`

- A “**"**” surrounds a double quoted flow scalar.

[18] `c-double-quote ::= “"”`

Example 4.8. Quoted Scalar Indicators

```
single: 'text'
double: "text"
```

```
Legend:
  c-single-quote
  c-double-quote
```

```
%YAML 1.1
---
!!map {
  ? !!str "double"
  : !!str "text",
  ? !!str "single"
  : !!str "text",
}
```

- A “**%**” denotes a directive line.

[19] `c-directive ::= “%”`

Example 4.9. Directive Indicator

```
%YAML 1.1
--- text
```

```
%YAML 1.1
---
!!str "text"
```

```
Legend:
  c-directive
```

- The “@” and “`” are *reserved* for future use.

```
[20] c-reserved ::= “@” | “`”
```

Example 4.10. Invalid use of Reserved Indicators

```
commercial-at: @text
grave-accent: `text
```

```
ERROR:
 Reserved indicators can't
 start a plain scalar.
```

- Any indicator character:

```
[21] c-indicator ::=   “-” | “?” | “:” | “,” | “[” | “]” | “{” | “}”
                     | “#” | “&” | “*” | “!” | “|” | “>” | “'” | “"”
                     | “%” | “@” | “`”
```

4.1.4. Line Break Characters

The Unicode standard defines the following *line break* characters:

```
[22] b-line-feed ::= #xA /*LF*/
[23] b-carriage-return ::= #xD /*CR*/
[24] b-next-line ::= #x85 /*NEL*/
[25] b-line-separator ::= #x2028 /*LS*/
[26] b-paragraph-separator ::= #x2029 /*PS*/
```

A YAML processor must accept all the possible Unicode line break characters.

```
[27] b-char ::=   b-line-feed | b-carriage-return | b-next-line
                | b-line-separator | b-paragraph-separator
```

Line breaks can be grouped into two categories. *Specific line breaks* have well-defined semantics for breaking text into lines and paragraphs, and must be preserved by the YAML processor inside scalar content.

```
[28] b-specific ::= b-line-separator | b-paragraph-separator
```

Generic line breaks do not carry a meaning beyond “ending a line”. Unlike specific line breaks, there are several widely used forms for generic line breaks.

```
[29] b-generic ::=   ( b-carriage-return b-line-feed ) /* DOS, Windows */
                   | b-carriage-return                 /* Macintosh */
                   | b-line-feed                       /* UNIX */
                   | b-next-line                       /* Unicode */
```

Generic line breaks inside scalar content must be *normalized* by the YAML processor. Each such line break must be parsed into a single line feed character. The original line break form is a presentation detail and must not be used to convey content information.

```
[30] b-as-line-feed ::= b-generic
[31] b-normalized ::= b-as-line-feed | b-specific
```

Normalization does not apply to ignored (escaped or chomped) generic line breaks.

```
[32] b-ignored-generic ::= b-generic
```

Outside scalar content, YAML allows any line break to be used to terminate lines.

```
[33] b-ignored-any ::= b-generic | b-specific
```

On output, a YAML processor is free to present line breaks using whatever convention is most appropriate, though specific line breaks must be preserved in scalar content. These rules are compatible with Unicode's newline guidelines [http://www.unicode.org/unicode/reports/tr13/].

In the examples, line break characters are displayed as follows: “↓” or no glyph for a generic line break, “⇓” for a line separator and “¶” for a paragraph separator.

Example 4.11. Line Break Characters

```
|
  Generic line break (no glyph)
  Generic line break (glyphed)↓
  Line separator⇓
  Paragraph separator¶
```

```
%YAML 1.1
--- !!str
"Generic line break (no glyph)\n\
 Generic line break (glyphed)\n\
 Line separator\u2028\
 Paragraph separator\u2029"
```

```
Legend:
  b-generic b-line-separator
  b-paragraph-separator
```

4.1.5. Miscellaneous Characters

The YAML syntax productions make use of the following character range definitions:

- A non-break character:

```
[34] nb-char ::= c-printable - b-char
```

- An ignored space character outside scalar content. Such spaces are used for indentation and separation between tokens. To maintain portability, *tab* characters must not be used in these cases, since different systems treat tabs differently. Note that most modern editors may be configured so that pressing the tab key results in the insertion of an appropriate number of spaces.

[35] `s-ignored-space ::= #x20 /*SP*/`

Example 4.12. Invalid Use of Tabs

```
# Tabs do's and don'ts:
# comment: →
quoted: "Quoted →"
block: |
  void main() {
  →printf("Hello, world!\n");
  }
elsewhere:→# separation
→indentation, in→plain scalar
```

```
ERROR:
 Tabs may appear inside
 comments and quoted or
 block scalar content.
 Tabs must not appear
 elsewhere, such as
 in indentation and
 separation spaces.
```

- A *white space* character in quoted or block scalar content:

[36] `s-white ::= #x9 /*TAB*/ | #x20 /*SP*/`

In the examples, tab characters are displayed as the glyph "→". Space characters are sometimes displayed as the glyph "·" for clarity.

Example 4.13. Tabs and Spaces

```
··"Text·containing···
··both·space·and→
··→tab→characters"
```

```
Legend:
  #x9 (TAB) #x20 (SP)
```

```
%YAML 1.1
--- !!str
"Text·containing·\
 both·space·and·\
 tab→characters"
```

- An ignored white space character inside scalar content:

[37] `s-ignored-white ::= s-white`

- A non space (and non-break) character:

[38] `ns-char ::= nb-char - s-white`

- A decimal digit for numbers:

[39] `ns-dec-digit ::= [#x30-#x39] /*0-9*/`

- A hexadecimal digit for escape sequences:

```
[40] ns-hex-digit ::= ns-dec-digit | [#x41-#x46] /*A-F*/ | [#x61-#x66] /*a-f*/
```

- An ASCII letter (alphabetic) character:

```
[41] ns-ascii-letter ::= [#x41-#x5A] /*A-Z*/ | [#x61-#x7A] /*a-z*/
```

- A word (alphanumeric) character for identifiers:

```
[42] ns-word-char ::= ns-dec-digit | ns-ascii-letter | “-”
```

- A URI character for tags, as specified in RFC2396 [http://www.ietf.org/rfc/rfc2396.txt] with the addition of the “**[**” and “**]**” for presenting IPv6 addresses as proposed in RFC2732 [http://www.ietf.org/rfc/rfc2732.txt]. A limited form of 8-bit *escaping* is available using the “**%**” character. By convention, URIs containing 16 and 32 bit Unicode characters are encoded in UTF-8, and then each octet is written as a separate character.

```
[43] ns-uri-char ::=   ns-word-char | “%” ns-hex-digit ns-hex-digit
                     | “;” | “/” | “?” | “:” | “@” | “&” | “=” | “+” | “$” | “,”
                     | “_” | “.” | “!” | “~” | “*” | “'” | “(” | “)” | “[” | “]”
```

- The “**!**” character is used to indicate the end of a named tag handle; hence its use in tag shorthands is restricted.

```
[44] ns-tag-char ::= ns-uri-char - “!”
```

4.1.6. Escape Sequences

All non-printable characters must be presented as *escape sequences*. Each escape sequences must be parsed into the appropriate Unicode character. The original escape sequence form is a presentation detail and must not be used to convey content information. YAML escape sequences use the “****” notation common to most modern computer languages. Note that escape sequences are only interpreted in double quoted scalars. In all other scalar styles, the “****” character has no special meaning and non-printable characters are not available.

```
[45] c-escape ::= “\”
```

YAML escape sequences are a superset of C's escape sequences:

- Escaped ASCII null (**#x0**) character:

```
[46] ns-esc-null ::= “\” “0”
```

- Escaped ASCII bell (**#x7**) character:

```
[47] ns-esc-bell ::= “\” “a”
```

- Escaped ASCII backspace (**#x8**) character:

```
[48] ns-esc-backspace ::= “\” “b”
```

- Escaped ASCII horizontal tab (**#x9**) character:

```
[49] ns-esc-horizontal-tab ::= “\” “t” | “\” #x9
```

- Escaped ASCII line feed (**`#xA`**) character:

[50] `ns-esc-line-feed ::= “\” “n”`

- Escaped ASCII vertical tab (**`#xB`**) character:

[51] `ns-esc-vertical-tab ::= “\” “v”`

- Escaped ASCII form feed (**`#xC`**) character:

[52] `ns-esc-form-feed ::= “\” “f”`

- Escaped ASCII carriage return (**`#xD`**) character:

[53] `ns-esc-carriage-return ::= “\” “r”`

- Escaped ASCII escape (**`#x1B`**) character:

[54] `ns-esc-escape ::= “\” “e”`

- Escaped ASCII space (**`#x20`**) character:

[55] `ns-esc-space ::= “\” #x20`

- Escaped ASCII double quote (“**`"`**”):

[56] `ns-esc-double-quote ::= “\” “"”`

- Escaped ASCII back slash (“**`\`**”):

[57] `ns-esc-backslash ::= “\” “\”`

- Escaped Unicode next line (**`#x85`**) character:

[58] `ns-esc-next-line ::= “\” “N”`

- Escaped Unicode non-breaking space (**`#xA0`**) character:

[59] `ns-esc-non-breaking-space ::= “\” “_”`

- Escaped Unicode line separator (**`#x2028`**) character:

[60] `ns-esc-line-separator ::= “\” “L”`

- Escaped Unicode paragraph separator (**`#x2029`**) character:

[61] `ns-esc-paragraph-separator ::= “\” “P”`

- Escaped 8-bit Unicode character:

[62] `ns-esc-8-bit ::= “\” “x” ( ns-hex-digit x 2 )`

- Escaped 16-bit Unicode character:

```
[63] ns-esc-16-bit ::= “\” “u” ( ns-hex-digit x 4 )
```

- Escaped 32-bit Unicode character:

```
[64] ns-esc-32-bit ::= “\” “U” ( ns-hex-digit x 8 )
```

- Any escaped character:

```
[65] ns-esc-char ::=   ns-esc-null | ns-esc-bell | ns-esc-backspace
                     | ns-esc-horizontal-tab | ns-esc-line-feed
                     | ns-esc-vertical-tab | ns-esc-form-feed
                     | ns-esc-carriage-return | ns-esc-escape | ns-esc-space
                     | ns-esc-double-quote | ns-esc-backslash
                     | ns-esc-next-line | ns-esc-non-breaking-space
                     | ns-esc-line-separator | ns-esc-paragraph-separator
                     | ns-esc-8-bit | ns-esc-16-bit | ns-esc-32-bit
```

Example 4.14. Escaped Characters

```
"Fun with \\
 \" \a \b \e \f \↓
 \n \r \t \v \0 \⇓
 \  \_ \N \L \P \¶
 \x41 \u0041 \U00000041"
```

```
%YAML 1.1
---
"Fun with \x5C
 \x22 \x07 \x08 \x1B \0C
 \x0A \x0D \x09 \x0B \x00
 \x20 \xA0 \x85 \u2028 \u2029
 A A A"
```

```
Legend:
  ns-esc-char
```

Example 4.15. Invalid Escaped Characters

```
Bad escapes:
  "\c
  \xq-"
```

```
ERROR:
- c is an invalid escaped character.
- q and - are invalid hex digits.
```

4.2. Syntax Primitives

4.2.1. Production Parameters

As YAML's syntax is designed for maximal readability, it makes heavy use of the context that each syntactical entity appears in. For notational compactness, this is expressed using parameterized BNF productions. The set of parameters and the range of allowed values depend on the specific production. The full list of possible parameters and their values is:

Indentation: `n` or `m` — Since the character stream depends upon indentation level to delineate blocks, many productions are parameterized by it. In some cases, the notations “**`production(<n)`**”, “**`production(≤n)`**”

and “**`production(>n)`**” are used; these are shorthands for “**`production(m)`**” for some specific `m` where 0 ≤ `m` < `n`, 0 ≤ `m` ≤ `n` and `m` > `n`, respectively.

Context: `c` — YAML supports two groups of *contexts*, distinguishing between block styles and flow styles. In the block styles, indentation is used to delineate structure. Due to the fact that the “**`-`**” character denoting a block sequence entry is perceived as an indentation character, some productions distinguish between the block-in context (inside a block sequence) and the block-out context (outside one). In the flow styles, explicit indicators are used to delineate structure. As plain scalars have no such indicators, they are the most context sensitive, distinguishing between being nested inside a flow collection (flow-in context) or being outside one (flow-out context). YAML also provides a terse and intuitive syntax for simple keys. Plain scalars in this (flow-key) context are the most restricted, for readability and implementation reasons.

(Scalar) Style: `s` — Scalar content may be presented in one of five styles: the plain, double quoted and single quoted flow styles, and the literal and folded block styles.

(Block) Chomping: `t` — Block scalars offer three possible mechanisms for chomping any trailing line breaks: strip, clip and keep.

4.2.2. Indentation Spaces

In a YAML character stream, structure is often determined from *indentation*, where indentation is defined as a line break character (or the start of the stream) followed by zero or more space characters. Note that indentation must not contain any tab characters. The amount of indentation is a presentation detail used exclusively to delineate structure and is otherwise ignored. In particular, indentation characters must never be considered part of a node's content information.

[66] `s-indent(n) ::= s-ignored-space x n`

Example 4.16. Indentation Spaces

```
··# Leading comment line spaces are
···# neither content nor indentation.
····
Not indented:
·By one space: |
····By four
······spaces
·Flow style: [    # Leading spaces
···By two,        # in flow style
··Also by two,    # are neither
··→Still by two   # content nor
····]             # indentation.
```

```
%YAML 1.1
---
!!map {
  ? !!str "Not indented"
  : !!map {
      ? !!str "By one space"
      : !!str "By four\n  spaces\n",
      ? !!str "Flow style"
      : !!seq [
          !!str "By two",
          !!str "Still by two",
          !!str "Again by two",
        ]
    }
}
```

```
Legend:
  s-indent(n) Content
  Neither content nor indentation
```

In general, a node must be indented further than its parent node. All sibling nodes must use the exact same indentation level, however the content of each sibling node may be further indented independently. The “**`-`**”, “**`?`**” and “**`:`**” characters used to denote block collection entries are perceived by people to be part of the indentation. Hence the indentation rules are slightly more flexible when dealing with these indicators. First, a block sequence need not be indented relative to its parent node, unless that node is also a block sequence. Second, compact in-line notations allow a nested collection to begin immediately following the indicator (where the indicator is counted as part of the indentation). This provides for an intuitive collection nesting syntax.

4.2.3. Comments

An explicit *comment* is is marked by a *"#" indicator*. Comments are a presentation detail and must have no effect on the serialization tree (and hence the representation graph).

```
[67] c-nb-comment-text ::= "#" nb-char*
```

Comments always span to the end of the line.

```
[68] c-b-comment ::= c-nb-comment-text? b-ignored-any
```

Outside scalar content, comments may appear on a line of their own, independent of the indentation level. Note that tab characters must not be used and that empty lines outside scalar content are taken to be (empty) comment lines.

```
[69] l-comment ::= s-ignored-space* c-b-comment
```

Example 4.17. Comment Lines

```
··# Comment↓
···↓
↓
```

```
# This stream contains no
# documents, only comments.
```

```
Legend:
  c-b-comment l-comment
```

When a comment follows another syntax element, it must be separated from it by space characters. Like the comment itself, such characters are not considered part of the content information.

```
[70] s-b-comment ::= ( s-ignored-space+ c-nb-comment-text )?
                     b-ignored-any
```

Example 4.18. Comments Ending a Line

```
key:····# Comment↓
  value↓
```

```
Legend:
  c-nb-comment-text s-b-comment
```

```
%YAML 1.1
---
!!map {
  ? !!str "key"
  : !!str "value"
}
```

In most cases, when a line may end with a comment, YAML allows it to be followed by additional comment lines.

```
[71] c-l-comments ::= c-b-comment l-comment*
[72] s-l-comments ::= s-b-comment l-comment*
```

Example 4.19. Multi-Line Comments

```
key:····# Comment↓
········# lines↓
  value↓
↓
```

```
%YAML 1.1
---
!!map {
  ? !!str "key"
  : !!str "value"
}
```

```
Legend:
  s-b-comment l-comment s-l-comments
```

4.2.4. Separation Spaces

Outside scalar content, YAML uses space characters for *separation* between tokens. Note that separation must not contain tab characters. Seperation spaces are a presentation detail used exclusively to delineate structure and are otherwise ignored; in particular, such characters must never be considered part of a node's content information.

```
[73] s-separate(n,c) ::= c = block-out ⇒ s-separate-lines(n)
                         c = block-in  ⇒ s-separate-lines(n)
                         c = flow-out  ⇒ s-separate-lines(n)
                         c = flow-in   ⇒ s-separate-lines(n)
                         c = flow-key  ⇒ s-separate-spaces
```

- YAML usually allows separation spaces to include a comment ending the line and additional comment lines. Note that the token following the separation comment lines must be properly indented, even though there is no such restriction on the separation comment lines themselves.

```
[74] s-separate-lines(n) ::=   s-ignored-space+
                             | ( s-l-comments s-indent(n) s-ignored-space* )
```

- Inside simple keys, however, separation spaces are confined to the current line.

```
[75] s-separate-spaces ::= s-ignored-space+
```

Example 4.20. Separation Spaces

```
{·first:·Sammy,·last:·Sosa·}:↓
# Statistics:
··hr:··# Home runs
····65
··avg:·# Average
····0.278
```

```
Legend:
  s-separate-spaces
  s-separate-lines(n)
  s-indent(n)
```

```
%YAML 1.1
---
!!map {
  ? !!map {
    ? !!str "first"
    : !!str "Sammy",
    ? !!str "last"
    : !!str "Sosa"
  }
  : !!map {
    ? !!str "hr"
    : !!int "65",
    ? !!str "avg"
    : !!float "0.278"
  }
}
```

4.2.5. Ignored Line Prefix

YAML discards the “empty” *prefix* of each scalar content line. This prefix always includes the indentation, and depending on the scalar style may also include all leading white space. The ignored prefix is a presentation detail and must never be considered part of a node's content information.

```
[76] s-ignored-prefix(n,s) ::= s = plain   ⇒ s-ignored-prefix-plain(n)
                               s = double  ⇒ s-ignored-prefix-quoted(n)
                               s = single  ⇒ s-ignored-prefix-quoted(n)
                               s = literal ⇒ s-ignored-prefix-block(n)
                               s = folded  ⇒ s-ignored-prefix-block(n)
```

- Plain scalars must not contain any tab characters, and all leading spaces are always discarded.

```
[77] s-ignored-prefix-plain(n) ::= s-indent(n) s-ignored-space*
```

- Quoted scalars may contain tab characters. Again, all leading white space is always discarded.

```
[78] s-ignored-prefix-quoted(n) ::= s-indent(n) s-ignored-white*
```

- Block scalars rely on indentation; hence leading white space, if any, is not discarded.

```
[79] s-ignored-prefix-block(n) ::= s-indent(n)
```

Example 4.21. Ignored Prefix

```
plain: text
··lines
quoted: "text
··→lines"
block: |
··text
···→lines
```

```
%YAML 1.1
---
!!map {
  ? !!str "plain"
  : !!str "text lines",
  ? !!str "quoted"
  : !!str "text lines",
  ? !!str "block"
  : !!str "text·→lines\n"
}
```

```
Legend:
  s-ignored-prefix-plain(n)
  s-ignored-prefix-quoted(n)
  s-ignored-prefix-block(n)
  s-indent(n)
```

An *empty line* line consists of the ignored prefix followed by a line break. When trailing block scalars, such lines can also be interpreted as (empty) comment lines. YAML provides a chomping mechanism to resolve this ambiguity.

```
[80] l-empty(n,s) ::= ( s-indent(<n) | s-ignored-prefix(n,s) )
                      b-normalized
```

Example 4.22. Empty Lines

```

- foo
·↓
  bar
- |-
  foo
·↓
  bar
··↓
```

```
%YAML 1.1
---
!!seq {
  !!str "foo\nbar",
  !!str "foo\n\nbar"
}
```

```
Legend:
  l-empty(n,s)
  l-comment
```

4.2.6. Line Folding

Line folding allows long lines to be broken for readability, while retaining the original semantics of a single long line. When folding is done, any line break ending an empty line is preserved. In addition, any specific line breaks are also preserved, even when ending a non-empty line.

```
[81] b-l-folded-specific(n,s) ::= b-specific l-empty(n,s)*
```

Hence, folding only applies to generic line breaks that end non-empty lines. If the following line is also not empty, the generic line break is converted to a single space (**`#x20`**).

```
[82] b-l-folded-as-space ::= b-generic
```

If the following line is empty line, the generic line break is ignored.

```
[83] b-l-folded-trimmed(n,s) ::= b-ignored-generic l-empty(n,s)+
```

Thus, a folded non-empty line may end with one of three possible folded line break forms. The original form of such a folded line break is a presentation detail and must not be used to convey node's content information.

```
[84] b-l-folded-any(n,s) ::=   b-l-folded-specific(n,s)
                             | b-l-folded-as-space
                             | b-l-folded-trimmed(n,s)
```

Example 4.23. Line Folding

```
>-
  specific⇓
  trimmed↓
··↓
·↓
↓
  as↓
  space
```

```
%YAML 1.1
--- !!str
"specific\L\
 trimmed\n\n\n\
 as space"
```

```
Legend:
  b-l-folded-specific(n,s)
  b-l-folded-as-space
  b-l-folded-trimmed(n,s)
```

The above rules are common to both the folded block style and the scalar flow styles. Folding does distinguish between the folded block style and the scalar flow styles in the following way:

Block Folding
: In the folded block style, folding does not apply to line breaks or empty lines that preced or follow a text line containing leading white space. Note that such a line may consist of only such leading white space; an empty block line is confined to (optional) indentation spaces only. Further, the final line break and empty lines are subject to chomping, and are never folded. The combined effect of these rules is that each “paragraph” is interpreted as a line, empty lines are used to present a line feed, the formatting of “more indented” lines is preserved, and final line breaks may be included or excluded from the node's content information as appropriate.

Flow Folding
: Folding in flow styles provides more relaxed, less powerful semantics. Flow styles typically depend on explicit indicators to convey structure, rather than indentation. Hence, in flow styles, spaces preceding or following the text in a line are a presentation detail and must not be considered a part of the node's content information. Once all such spaces have been discarded, folding proceeds as described above. In contrast with the block folded style, all line breaks are folded, without exception, and a line consisting only of spaces is considered to be an empty line, regardless of the number of spaces. The combined effect of these processing rules is that each “paragraph” is interpreted as a line, empty lines are used to present a line feed, and text can be freely “more indented” without affecting the node's content information.

4.3. YAML Character Stream

A YAML character stream may contain several YAML documents, denoted by document boundary markers. Each document presents a single independent root node and may be preceded by a series of directives.

4.3.1. Directives

Directives are instructions to the YAML processor. Like comments, directives are presentation details and are not reflected in the serialization tree (and hence the representation graph). This specification defines two directives, "**`YAML`**" and "**`TAG`**", and *reserves* all other directives for future use. There is no way to define private directives. This is intentional.

```
[85] l-directive ::= l-yaml-directive | l-tag-directive | l-reserved-directive
```

Each directive is specified on a separate non-indented line starting with the *"**`%`**" indicator*, followed by the directive name and a space-separated list of parameters. The semantics of these tokens depend on the specific directive. A YAML processor should ignore unknown directives with an appropriate warning.

```
[86] l-reserved-directive ::= "%" ns-directive-name
                              ( s-ignored-space+ ns-directive-parameter )*
                              s-l-comments
[87] ns-directive-name ::= ns-char+
[88] ns-directive-parameter ::= ns-char+
```

Example 4.24. Reserved Directives

```
%FOO  bar baz # Should be ignored
              # with a warning.
--- "foo"
```

```
%YAML 1.1
--- !!str
"foo"
```

```
Legend:
  l-reserved-directive
  ns-directive-name
  ns-directive-parameter
```

4.3.1.1. "YAML" Directive

The *"**`YAML`**" directive* specifies the version of YAML the document adheres to. This specification defines version "**`1.1`**". A version 1.1 YAML processor should accept documents with an explicit "**`%YAML 1.1`**" directive, as well as documents lacking a "**`YAML`**" directive. Documents with a "**`YAML`**" directive specifying a higher minor version (e.g. "**`%YAML 1.2`**") should be processed with an appropriate warning. Documents with a "**`YAML`**" directive specifying a higher major version (e.g. "**`%YAML 2.0`**") should be rejected with an appropriate error message.

```
[89] l-yaml-directive ::= "%" "Y" "A" "M" "L"
                          s-ignored-space+ ns-yaml-version
                          s-l-comments
[90] ns-yaml-version ::= ns-dec-digit+ "." ns-dec-digit+
```

Example 4.25. "YAML" directive

```
%YAML 1.2 # Attempt parsing
          # with a warning
---
"foo"
```

```
%YAML 1.1
---
!!str "foo"
```

```
Legend:
  l-yaml-directive  ns-yaml-version
```

It is an error to specify more than one “**YAML**” directive for the same document, even if both occurences give the same version number.

Example 4.26. Invalid Repeated YAML directive

```
%YAML 1.1
%YAML 1.1
foo
```

```
ERROR:
The YAML directive must only be
given at most once per document.
```

4.3.1.2. “TAG” Directive

The “***TAG***” *directive* establishes a shorthand notation for specifying node tags. Each “**TAG**” directive associates a handle with a prefix, allowing for compact and readable tag notation.

```
[91]  l-tag-directive ::= “%” “T” “A” “G”
                          s-ignored-space+ c-tag-handle
                          s-ignored-space+ ns-tag-prefix
                          s-l-comments
```

Example 4.27. “TAG” directive

```
%TAG !yaml! tag:yaml.org,2002:↓
---
!yaml!str "foo"
```

```
%YAML 1.1
---
!!str "foo"
```

```
Legend:
  l-tag-directive
  c-tag-handle ns-tag-prefix
```

It is an error to specify more than one “**TAG**” directive for the same handle in the same document, even if both occurences give the same prefix.

Example 4.28. Invalid Repeated TAG directive

```
%TAG ! !foo
%TAG ! !foo
bar
```

```
ERROR:
The TAG directive must only
be given at most once per
handle in the same document.
```

4.3.1.2.1. Tag Prefixes

There are two *tag prefix* variants:

```
[92]  ns-tag-prefix ::= ns-local-tag-prefix | ns-global-tag-prefix
```

Local Tags — If the prefix begins with a "**!**" character, shorthands using the handle are expanded to a local tag beginning with "**!**". Note that such a tag is intentionally not a valid URI, since its semantics are specific to the application. In particular, two documents in the same stream may assign different semantics to the same local tag.

```
[93] ns-local-tag-prefix ::= "!" ns-uri-char*
```

Global Tags — If the prefix begins with a character other than "**!**", it must to be a valid URI prefix, and should contain at least the scheme and the authority. Shorthands using the associated handle are expanded to globally unique URI tags, and their semantics is consistent across applications. In particular, two documents in different streams must assign the same semantics to the same global tag.

```
[94] ns-global-tag-prefix ::= ns-tag-char ns-uri-char*
```

Example 4.29. Tag Prefixes

```
%TAG !       !foo
%TAG !yaml! tag:yaml.org,2002:
---
- !bar "baz"
- !yaml!str "string"
```

```
%YAML 1.1
---
!!seq [
  !<!foobar> "bar",
  !<tag:yaml.org,2002:str> "string"
]
```

```
Legend:
  ns-local-tag-prefix  ns-global-tag-prefix
```

4.3.1.2.2. Tag Handles

The *tag handle* exactly matches the prefix of the affected shorthand. There are three tag handle variants:

```
[95] c-tag-handle ::=   c-primary-tag-handle
                      | ns-secondary-tag-handle
                      | c-named-tag-handle
```

Primary Handle — The *primary tag handle* is a single "**!**" character. This allows using the most compact possible notation for a single "primary" name space. By default, the prefix associated with this handle is "**!**". Thus, by default, shorthands using this handle are interpreted as local tags. It is possible to override this behavior by providing an explicit "**TAG**" directive associating a different prefix for this handle. This provides smooth migration from using local tags to using global tags by a simple addition of a single "**TAG**" directive.

```
[96] c-primary-tag-handle ::= "!"
```

Example 4.30. Migrating from Local to Global Tags

```
# Private application:
!foo "bar"
```

```
%YAML 1.1
---
!<!foo> "bar"
```

```
# Migrated to global:
%TAG ! tag:ben-kiki.org,2000:app/
---
!foo "bar"
```

```
%YAML 1.1
---
!<tag:ben-kiki.org,2000:app/foo> "bar"
```

Secondary Handle — The *secondary tag handle* is written as “**`!!`**”. This allows using a compact notation for a single “secondary” name space. By default, the prefix associated with this handle is “**`tag:yaml.org,2002:`**” used by the YAML tag repository [http://yaml.org/type/index.html] providing recommended tags for increasing the portability of YAML documents between different applications. It is possible to override this behavior by providing an explicit “**`TAG`**” directive associating a different prefix for this handle.

[97] `ns-secondary-tag-handle ::= “!” “!”`

Named Handles — A *named tag handle* surrounds the non-empty name with “**`!`**” characters. A handle name must only be used in a shorthand if an explicit “**`TAG`**” directive has associated some prefix with it. The name of the handle is a presentation detail and is not part of the node's content information. In particular, the YAML processor need not preserve the handle name once parsing is completed.

[98] `c-named-tag-handle ::= “!” ns-word-char+ “!”`

Example 4.31. Tag Handles

```
# Explicitly specify default settings:
%TAG !     !
%TAG !!    tag:yaml.org,2002:
# Named handles have no default:
%TAG !o! tag:ben-kiki.org,2000:
---
- !foo "bar"
- !!str "string"
- !o!type "baz"
```

```
%YAML 1.1
---
!!seq [
  !<!foo> "bar",
  !<tag:yaml.org,2002:str> "string"
  !<tag:ben-kiki.org,2000:type> "baz"
]
```

```
Legend:
  c-primary-tag-handle
  c-secondary-tag-handle
  c-named-tag-handle
```

4.3.2. Document Boundary Markers

YAML streams use *document boundary markers* to allow more than one document to be contained in the same stream. Such markers are a presentation detail and are used exclusively to convey structure. A line beginning with “**`---`**” may be used to explicitly denote the beginning of a new YAML document.

[99] `c-document-start ::= “-” “-” “-”`

When YAML is used as the format of a communication channel, it is useful to be able to indicate the end of a document without closing the stream, independent of starting the next document. Lacking such a marker, the YAML processor reading the stream would be forced to wait for the header of the next document (that may be long time in coming) in order to detect the end of the previous one. To support this scenario, a YAML document may be terminated by an explicit end line denoted by "`...`", followed by optional comments. To ease the task of concatenating YAML streams, the end marker may be repeated.

```
[100] c-document-end ::= "." "." "."
[101] l-document-suffix ::= ( c-document-end s-l-comments )+
```

Example 4.32. Document Boundary Markers

```
---↓
foo
...
# Repeated end marker.
...↓
---↓
bar
# No end marker.
---↓
baz
...↓
```

```
%YAML 1.1
---
!!str "foo"
%YAML 1.1
---
!!str "bar"
%YAML 1.1
---
!!str "baz"
```

```
Legend:
  c-document-start  l-document-suffix
```

4.3.3. Documents

A YAML *document* is a single native data structure presented as a single root node. Presentation details such as directives, comments, indentation and styles are not considered part of the content information of the document.

Explicit Documents An *explicit document* begins with a document start marker followed by the presentation of the root node. The node may begin in the same line as the document start marker. If the explicit document's node is completely empty, it is assumed to be an empty plain scalar with no specified properties. Optional document end marker(s) may follow the document.

```
[102] l-explicit-document ::= c-document-start
                              ( s-l+block-node(-1,block-in) | s-l-empty-block )
                              l-document-suffix?
```

Implicit Documents An *implicit document* does not begin with a document start marker. In this case, the root node must not be presented as a completely empty node. Again, optional document end marker(s) may follow the document.

```
[103] l-implicit-document ::= s-ignored-space* ns-l+block-node(-1,block-in)
                              l-document-suffix?
```

In general, the document's node is indented as if it has a parent indented at -1 spaces. Since a node must be more indented that its parent node, this allows the document's node to be indented at zero or more spaces. Note that flow scalar continuation lines must be indented by at least one space, even if their first line is not indented.

Example 4.33. Documents

```
"Root flow
 scalar"
--- !!str >
 Root block
 scalar
---
# Root collection:
foo : bar
... # Is optional.
---
# Explicit document may be empty.
```

```
%YAML 1.1
---
!!str "Root flow scalar"
%YAML 1.1
---
!!str "Root block scalar"
%YAML 1.1
---
!!map {
  ? !!str "foo"
  : !!str "bar"
}
---
!!str ""
```

```
Legend:
  l-implicit-document  l-explicit-document
```

4.3.4. Complete Stream

A sequence of bytes is a YAML character *stream* if, taken as a whole, it complies with the **`l-yaml-stream`** production. The stream begins with a prefix containing an optional byte order mark denoting its character encoding, followed by optional comments. Note that the stream may contain no documents, even if it contains a non-empty prefix. In particular, a stream containing no chareacters is valid and contains no documents.

```
[104] l-yaml-stream ::= c-byte-order-mark? l-comment*
                        ( l-first-document l-next-document* )?
```

Example 4.34. Empty Stream

```
⇔# A stream may contain
# no documents.
```

```
# This stream contains no
# documents, only comments.
```

```
Legend:
  l-yaml-stream
```

The first document may be implicit (omit the document start marker). In such a case it must not specify any directives and will be parsed using the default settings. If the document is explicit (begins with an document start marker), it may specify directives to control its parsing.

```
[105] l-first-document ::= ( l-implicit-document
                           | ( l-directive* l-explicit-document ) )
```

Example 4.35. First Document

```
# Implicit document. Root
# collection (mapping) node.
foo : bar
```

```
%YAML 1.1
---
!!str "Text content\n"
```

```
# Explicit document. Root
# scalar (literal) node.
--- |
 Text content
```

```
%YAML 1.1
---
!!map {
  ? !!str "foo"
  : !!str "bar"
}
```

```
Legend:
  l-first-document
```

To ease the task of concatenating character streams, following documents may begin with a byte order mark and comments, though the same character encoding must be used through the stream. Each following document must be explicit (begin with a document start marker). If the document specifies no directives, it is parsed using the same settings as the previous document. If the document does specify any directives, all directives of previous documents, if any, are ignored.

```
[106] l-next-document ::= c-byte-order-mark? l-comment*
                          l-directive* l-explicit-document
```

Example 4.36. Next Documents

```
! "First document"
---
!foo "No directives"
%TAG ! !foo
---
!bar "With directives"
%YAML 1.1
---
!baz "Reset settings"
```

```
%YAML 1.1
---
!!str "First document"
---
!<!foo> "No directives"
---
!<!foobar> "With directives"
---
!<!baz> "Reset settings"
```

```
Legend:
  l-next-document
```

4.4. Nodes

Each *presentation node* may have two optional *properties*, anchor and tag, in addition to its content. Node properties may be specified in any order before the node's content, and either or both may be omitted from the character stream.

```
[107] c-ns-properties(n,c) ::=   ( c-ns-tag-property
                                   ( s-separate(n,c) c-ns-anchor-property )? )
                               | ( c-ns-anchor-property
                                   ( s-separate(n,c) c-ns-tag-property )? )
```

Example 4.37. Node Properties

```
!!str
 &a1↓
  "foo" : !!str bar
&a2 baz : *a1
```

```
Legend:
  c-ns-anchor-property c-ns-tag-property
  c-ns-properties(n,c)
```

```
%YAML 1.1
---
!!map {
  ? &A1 !!str "foo"
  : !!str "bar",
  ? !!str &A2 "baz"
  : *a1
}
```

4.4.1. Node Anchors

The *anchor property* marks a node for future reference. An anchor is denoted by the *“**&**” indicator*. An alias node can then be used to indicate additional inclusions of the anchored node by specifying its anchor. An anchored node need not be referenced by any alias node; in particular, it is valid for all nodes to be anchored.

[108] `c-ns-anchor-property ::= “&” ns-anchor-name`

Note that as a serialization detail, the anchor name is preserved in the serialization tree. However, it is not reflected in the representation graph and must not be used to convey content information. In particular, the YAML processor need not preserve the anchor name once the representation is composed.

[109] `ns-anchor-name ::= ns-char+`

Example 4.38. Node Anchors

```
First occurence: &anchor Value
Second occurence: *anchor
```

```
Legend:
  c-ns-anchor-property
  ns-anchor-name
```

```
%YAML 1.1
---
!!map {
  ? !!str "First occurence"
  : &A !!str "Value",
  ? !!str "Second occurence"
  : *A
}
```

4.4.2. Node Tags

The *tag property* identifies the type of the native data structure presented by the node. A tag is denoted by the *“**!**” indicator*. In contrast with anchors, tags are an inherent part of the representation graph.

```
[110] c-ns-tag-property ::=   c-verbatim-tag | c-ns-shorthand-tag
                            | c-ns-non-specific-tag
```

Verbatim Tags

A tag may be written *verbatim* by surrounding it with the "**<**" *and* "**>**" characters. In this case, the YAML processor must deliver the verbatim tag as-is to the application. In particular, verbatim tags are not subject to tag resolution. A verbatim tag must either begin with a "**!**" (a local tag) or be a valid URI (a global tag).

```
[111] c-verbatim-tag ::= "!" "<" ns-uri-char+ ">"
```

Example 4.39. Verbatim Tags

```
!<tag:yaml.org,2002:str> foo :
  !<!bar> baz
```

```
Legend:
  c-verbatim-tag
```

```
%YAML 1.1
---
!!map {
  ? !<tag:yaml.org,2002:str> "foo"
  : !<!bar> "baz"
}
```

Example 4.40. Invalid Verbatim Tags

```
- !<!> foo
- !<$:?> bar
```

```
ERROR:
- Verbatim tags aren't resolved,
  so ! is invalid.
- The $:? tag is neither a global
  URI tag nor a local tag starting
  with !.
```

Tag Shorthands

A *tag shorthand* consists of a valid tag handle followed by a non-empty suffix. The tag handle must be associated with a prefix, either by default or by using a "**TAG**" directive. The resulting parsed tag is the concatenation of the prefix and the suffix, and must either begin with "**!**" (a local tag) or be a valid URI (a global tag). When the primary tag handle is used, the suffix must not contain any "**!**" character, as this would cause the tag shorthand to be interpreted as having a named tag handle. If the "**!**" character exists in the suffix of a tag using the primary tag handle, it must be escaped as "**%21**", and the parser should expand this particular escape sequence before passing the tag to the application. This behavior is consistent with the URI character quoting rules (specifically, section 2.3 of RFC2396 [http://www.ietf.org/rfc/rfc2396.txt]), and ensures the choice of tag handle remains a presentation detail and is not reflected in the serialization tree (and hence the representation graph). In particular, the tag handle may be discarded once parsing is completed.

```
[112] c-ns-shorthand-tag ::=   ( c-primary-tag-handle ns-tag-char+ )
                             | ( ns-secondary-tag-handle ns-uri-char+ )
                             | ( c-named-tag-handle ns-uri-char+ )
```

Example 4.41. Tag Shorthands

```
%TAG !o! tag:ben-kiki.org,2000:
---
- !local foo
- !!str bar
- !o!type baz
```

```
Legend:
  c-ns-shorthand-tag
```

```
%YAML 1.1
---
!!seq [
  !<!local> "foo",
  !<tag:yaml.org,2002:str> "bar",
  !<tag:ben-kiki.org,2000:type> "baz",
]
```

Example 4.42. Invalid Shorthand Tags

```
%TAG !o! tag:ben-kiki.org,2000:
---
- !$a!b foo
- !o! bar
- !h!type baz
```

```
ERROR:
- The !$a! looks like a handle.
- The !o! handle has no suffix.
- The !h! handle wasn't declared.
```

Non-Specific Tags If a node has no tag property, it is assigned a non-specific tag: “**?**” for plain scalars and “**!**” for all other nodes. Non-specific tags must be resolved to a specific tag for a complete representation graph to be composed. It is also possible for the tag property to explicitly specify the node has the “**!**” non-specific tag. This is only useful for plain scalars, causing them to be resolved as if they were non-plain (hence, by the common tag resolution convention, as “**`tag:yaml.org,2002:str`**”). There is no way to explicitly set the tag to the “**?**” non-specific tag. This is intentional.

[113] `c-ns-non-specific-tag ::= “!”`

Example 4.43. Non-Specific Tags

```
# Assuming conventional resolution:
- "12"
- 12
- ! 12
```

```
Legend:
  c-ns-non-specific-tag
```

```
%YAML 1.1
---
!!seq [
  !<tag:yaml.org,2002:str> "12",
  !<tag:yaml.org,2002:int> "12",
  !<tag:yaml.org,2002:str> "12",
]
```

4.4.3. Node Content

Node content may be presented in either a flow style or a block style. Block content always extends to the end of a line and uses indentation to denote structure, while flow content starts and ends at some non-space character within a line and uses indicators to denote structure. Each collection kind can be presented in a single flow collection style or a single block collection style. However, each collection kind also provides compact in-line forms for common cases. Scalar content may be presented in the plain style or one of the two quoted styles (the single quoted style and the double quoted style). Regardless of style, scalar content must always be indented by at least one space. In contrast, collection content need not be indented (note that the indentation of the first flow scalar line is determined by the block collection it is nested in, if any).

```
[114] ns-flow-scalar(n,c) ::=   c-plain(max(n,1),c)
                              | c-single-quoted(max(n,1),c)
                              | c-double-quoted(max(n,1),c)
[115] c-flow-collection(n,c) ::= c-flow-sequence(n,c) | c-flow-mapping(n,c)
[116] ns-flow-content(n,c) ::= ns-flow-scalar(n,c) | c-flow-collection(n,c)
[117] c-l+block-scalar(n) ::= c-l+folded(max(n,0)) | c-l+literal(max(n,0))
[118] c-l-block-collection(n,c) ::= c-l-block-sequence(n,c) | c-l-block-mapping(n)
[119] c-l+block-content(n,c) ::=   c-l+block-scalar(n)
                                 | c-l-block-collection(>n,c)
```

Example 4.44. Mandatory Scalar Indentation

```
---
foo:
·"bar
·baz"
---
"foo
·bar"
---
foo
·bar
--- |
·foo
...
```

```
%YAML 1.1
---
!!map {
  ? !!str "foo"
  : !!str "bar baz"
}
%YAML 1.1
---
!!str "foo bar"
%YAML 1.1
---
!!str "foo bar"
%YAML 1.1
---
!!str "foo bar\n"
```

```
Legend:
  Normal "more-indented" indentation
  Mandatory for "non-indented" scalar
```

Example 4.45. Flow Content

```
---
scalars:
  plain: !!str some text↓
  quoted:
    single: 'some text'↓
    double: "some text"↓
collections:
  sequence: !!seq [ !str entry,
    # Mapping entry:↓
      key: value ]↓
  mapping: { key: value }↓
```

```
%YAML 1.1
--- !!map {
  ? !!str "scalars" : !!map {
      ? !!str "plain"
      : !!str "some text",
      ? !!str "quoted"
      : !!map {
        ? !!str "single"
        : !!str "some text",
        ? !!str "double"
        : !!str "some text"
  } },
  ? !!str "collections": : !!map {
    ? !!str "sequence" : !!seq [
      ? !!str "entry",
      : !!map {
        ? !!str "key" : !!str "value"
    } ],
    ? !!str "mapping": : !!map {
      ? !!str "key" : !!str "value"
} } }
```

```
Legend:
  ns-flow-scalar
  c-flow-collection
  not content
```

Example 4.46. Block Content

```
block styles:
  scalars:
    literal: !!str |
      #!/usr/bin/perl
      print "Hello, world!\n";↓
    folded: >
      This sentence
      is false.↓
  collections: !!seq
    sequence: !!seq # Entry:↓
      - entry
      # Mapping entry:↓
      - key: value↓
    mapping: ↓
      key: value↓
```

```
%YAML 1.1
---
!!map {
  ? !!str "block styles" : !!map {
    ? !!str "scalars" : !!map {
      ? !!str "literal"
      : !!str "#!/usr/bin/perl\n\
          print \"Hello,
          world!!\\n\";\n",
      ? !!str "folded"
      : !!str "This sentence
          is false.\n"
    },
    ? !!str "collections" : !!map {
      ? !!str "sequence" : !!seq [
        !!str "entry",
        !!map {
          ? !!str "key" : !!str "value"
        }
      ],
      ? !!str "mapping" : !!map {
        ? !!str "key" : !!str "value"
} } } }
```

```
Legend:
  c-l+block-scalar
  c-l-block-collection
  not content
```

4.4.4. Alias Nodes

Subsequent occurrences of a previously serialized node are presented as *alias nodes*, denoted by the *“*****” indicator*. The first occurrence of the node must be marked by an anchor to allow subsequent occurrences to be presented as alias nodes. An alias node refers to the most recent preceding node having the same anchor. It is an error to have an alias node use an anchor that does not previously occur in the document. It is not an error to specify an anchor that is not used by any alias node. Note that an alias node must not specify any properties or content, as these were already specified at the first occurrence of the node.

```
[120] c-ns-alias-node ::= “*” ns-anchor-name
```

Example 4.47. Alias Nodes

```
First occurence: &anchor Value
Second occurence: *anchor
```

```
Legend:
  c-ns-alias-node
  ns-anchor-name
```

```
%YAML 1.1
---
!!map {
  ? !!str "First occurence"
  : &A !!str "Value",
  ? !!str "Second occurence"
  : *A
}
```

4.4.5. Complete Nodes

4.4.5.1. Flow Nodes

A complete *flow node* is either an alias node presenting a second occurence of a previous node, or consists of the node properties followed by the node's content. A node with empty content is considered to be an empty plain scalar.

```
[121] ns-flow-node(n,c) ::=   c-ns-alias-node | ns-flow-content(n,c)
                            | ( c-ns-properties(n,c)
                                ( /* empty plain scalar content */
                                | ( s-separate(n,c) ns-flow-content(n,c) ) ) )
```

Example 4.48. Flow Nodes in Flow Context

```
[
  Without properties,
  &anchor "Anchored",
  !!str 'Tagged',
  *anchor, # Alias node
  !!str,   # Empty plain scalar
]
```

```
%YAML 1.1
---
!!seq [
  !!str "Without properties",
  &A !!str "Anchored",
  !!str "Tagged",
  *A,
  !!str "",
]
```

```
Legend:
  ns-flow-node(n,c) ns-flow-content(n,c)
```

Since both the node's properties and node content are optional, this allows for a *completely empty node*. Completely empty nodes are only valid when following some explicit indicator for their existance.

```
[122] e-empty-flow ::= /* empty plain scalar node */
```

In the examples, completely empty nodes are displayed as the glyph “°”. Note that this glyph corresponds to a position in the characters stream rather than to an actual character.

Example 4.49. Completely Empty Flow Nodes

```
{
  ? foo :°,
  ?° : bar,
  ?° :°,
]
```

```
Legend:
  e-empty-flow
```

```
%YAML 1.1
---
!!map {
  ? !!str "foo"
  : !!str "",
  ? !!str "",
  : !!str "bar",
  ? !!str "",
  : !!str ""
}
```

4.4.5.2. Block Nodes

A complete *block node* consists of the node's properties followed by the node's content. In addition, a block node may consist of a (possibly completely empty) flow node followed by a line break (with optional comments).

```
[123] ns-l+flow-in-block(n,c) ::= ns-flow-node(n+1,flow-out) s-l-comments
[124] ns-l+block-in-block(n,c) ::= ( c-ns-properties(n+1,c) s-separate(n+1,c) )?
                                   c-l+block-content(n,c)
[125] ns-l+block-node(n,c) ::=   ns-l+block-in-block(n,c)
                               | ns-l+flow-in-block(n,c)
[126] s-l+block-node(n,c) ::= s-separate(n+1,c) ns-l+block-node(n,c)
```

Example 4.50. Block Nodes

```
-·"flow in block"↓
-·>
 Block scalar↓
-·!!map # Block collection
  foo : bar↓
```

```
Legend:
  ns-l+flow-in-block(n,c)
  ns-l+block-in-block(n,c)
  s-l+block-node(n,c)
```

```
%YAML 1.1
---
!!seq [
  !!str "",
  !!map {
    ? !!str "foo"
    : !!str "",
    ? !!str "",
    : !!str "bar",
    ? !!str "",
    : !!str ""
  }
]
```

A block node always spans to the end of the line, even when completely empty. Completely empty block nodes may only appear when there is some explicit indicator for their existance.

```
[127] s-l-empty-block ::= e-empty-flow s-l-comments
```

Example 4.51. Completely Empty Block Nodes

```
seq:
-° # Empty plain scalar↓
- ? foo
  :°↓
  ?°↓
  : bar,
  ?°↓
  :°↓
```

```
Legend:
  s-l-empty-block
```

```
%YAML 1.1
---
!!seq [
  !!str "",
  !!map {
    ? !!str "foo"
    : !!str "",
    ? !!str "",
    : !!str "bar",
    ? !!str "",
    : !!str ""
  }
]
```

4.5. Scalar Styles

YAML provides a rich set of *scalar styles* to choose from, depending upon the readability requirements: three scalar flow styles (the plain style and the two *quoted styles*: single quoted and double quoted), and two scalar block styles (the literal style and the folded style). Comments may precede or follow scalar content, but must not appear inside it. Scalar node style is a presentation detail and must not be used to convey content information, with the exception that untagged plain scalars are resolved in a distinct way.

4.5.1. Flow Scalar Styles

All *flow scalar styles* may span multiple lines, except when used in simple keys. Flow scalars are subject to (flow) line folding. This allows flow scalar content to be broken anywhere a single space character (**`#x20`**) separates non-space characters, at the cost of requiring an empty line to present each line feed character.

4.5.1.1. Double Quoted

The *double quoted style* is specified by surrounding *“**`"`**” indicators*. This is the only scalar style capable of expressing arbitrary strings, by using “**`\`**” escape sequences. Therefore, the “**`\`**” and “**`"`**” characters must also be escaped when present in double quoted content. Note it is an error for double quoted content to contain invalid escape sequences.

```
[128] nb-double-char ::= ( nb-char - “\” - “"” ) | ns-esc-char
[129] ns-double-char ::= nb-double-char - s-white
```

Double quoted scalars are restricted to a single line when contained inside a simple key.

```
[130] c-double-quoted(n,c) ::= “"” nb-double-text(n,c) “"”
[131] nb-double-text(n,c) ::= c = flow-out ⇒ nb-double-any(n)
                              c = flow-in  ⇒ nb-double-any(n)
                              c = flow-key ⇒ nb-double-single
[132] nb-double-any(n) ::= nb-double-single | nb-double-multi(n)
```

Example 4.52. Double Quoted Scalars

```
"simple key" : {
  "also simple" : value,
  ? "not a
  simple key" : "any
  value"
}
```

```
Legend:
  nb-double-single  nb-double-multi(n)
  c-double-quoted(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "simple key"
  : !!map {
    ? !!str "also simple"
    : !!str "value",
    ? !!str "not a simple key"
    : !!str "any value"
  }
}
```

A single line double quoted scalar is a sequence of (possibly escaped) non-break Unicode characters. All characters are considered content, including any leading or trailing white space characters.

```
[133] nb-double-single ::= nb-double-char*
```

In a multi-line double quoted scalar, line breaks are subject to flow line folding, and any trailing white space is excluded from the content. However, an *escaped line break* (using a “\”) is excluded from the content, while white space preceding it is preserved. This allows double quoted content to be broken at arbitrary positions.

```
[134] s-l-double-folded(n) ::= s-ignored-white* b-l-folded-any(n,double)
[135] s-l-double-escaped(n) ::= s-white* “\” b-ignored-any
                                l-empty(n,double)*
[136] s-l-double-break(n) ::= s-l-double-folded(n) | s-l-double-escaped(n)
```

Example 4.53. Double Quoted Line Breaks

```
 "as space→↓
 trimmed·↓
↓
 specific⇓
↓
 escaped→\¶
·↓
 none"
```

```
%YAML 1.1
---
!!str "as space \
  trimmed\n\
  specific\L\n\
  escaped\t\
  none"
```

```
Legend:
  s-l-double-folded(n)  s-l-double-escaped(n)
  s-ignored-white       s-white (Content)
```

A multi-line double quoted scalar consists of a (possibly empty) first line, any number of inner lines, and a final (possibly empty) last line.

```
[137] nb-double-multi(n) ::= nb-l-double-first(n)
                             l-double-inner(n)*
                             s-nb-double-last(n)
```

Leading white space in the first line is considered content only if followed by a non-space character or an escaped (ignored) line break.

```
[138] nb-l-double-first(n) ::= ( nb-double-char* ns-double-char )?
                               s-l-double-break(n)
```

Example 4.54. First Double Quoted Line

```
- "↓
  last"
- "·→↓
  last"
- "·→first↓
  last"
```

```
%YAML 1.1
---
!!seq [
  !!str " last",
  !!str " last",
  !!str " \tfirst last",
]
```

```
Legend:
  nb-l-double-first(n) s-ignored-white
```

All leading and trailing white space of an inner lines are excluded from the content. Note that such while prefix white space may contain tab characters, line indentation is restricted to space characters only. It is possible to force considering leading white space as content by escaping the first character ("`\·`", "`\→`" or "`\t`").

```
[139] l-double-inner(n) ::= s-ignored-prefix(n,double) ns-double-char
                            ( nb-double-char* ns-double-char )?
                            s-l-double-break(n)
```

Example 4.55. Inner Double Quoted Lines

```
 "first
·→inner 1→↓
·\·inner 2·\↓
 last"
```

```
%YAML 1.1
---
!!str "first \
  inner 1  \
  inner 2 \
  last"
```

```
Legend:
  l-double-inner(n)
  s-ignored-prefix(n,s) s-l-double-break(n)
```

The leading prefix white space of the last line is stripped in the same way as for inner lines. Trailing white space is considered content only if preceded by a non-space character.

```
[140] s-nb-double-last(n) ::= s-ignored-prefix(n,double)
                              ( ns-double-char nb-double-char* )?
```

Example 4.56. Last Double Quoted Line

```
- "first
··→"
- "first

··→last"
- "first
 inner
·\·→last"
```

```
%YAML 1.1
---
!!seq [
  !!str "first ",
  !!str "first\nlast",
  !!str "first inner  \tlast",
]
```

```
Legend:
  s-nb-double-last(n)
  s-ignored-prefix(n,s)
```

4.5.1.2. Single Quoted

The *single quoted style* is specified by surrounding "**'**" *indicators*. Therefore, within a single quoted scalar such characters need to be repeated. This is the only form of *escaping* performed in single quoted scalars. In particular, the "\" and "**"**" characters may be freely used. This restricts single quoted scalars to printable characters.

```
[141] c-quoted-quote ::= “'” “'”
[142] nb-single-char ::= ( nb-char - “'” ) | c-quoted-quote
[143] ns-single-char ::= nb-single-char - s-white
```

Example 4.57. Single Quoted Quotes

```
 'here''s to "quotes"'
```

```
Legend:
  single-quoted-quote
```

```
%YAML 1.1
---
!!str "here's to \"quotes\""
```

Single quoted scalars are restricted to a single line when contained inside a simple key.

```
[144] c-single-quoted(n,c) ::= “'” nb-single-text(n,c) “'”
[145] nb-single-text(n,c) ::= c = flow-out ⇒ nb-single-any(n)
                              c = flow-in  ⇒ nb-single-any(n)
                              c = flow-key ⇒ nb-single-single(n)
[146] nb-single-any(n) ::= nb-single-single(n) | nb-single-multi(n)
```

Example 4.58. Single Quoted Scalars

```
'simple key' : {
  'also simple' : value,
  ? 'not a
 simple key' : 'any
 value'
}
```

```
Legend:
  nb-single-single  nb-single-multi(n)
  c-single-quoted(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "simple key"
  : !!map {
    ? !!str "also simple"
    : !!str "value",
    ? !!str "not a simple key"
    : !!str "any value"
  }
}
```

A single line single quoted scalar is a sequence of non-break printable characters. All characters are considered content, including any leading or trailing white space characters.

[147] `nb-single-single(n) ::= nb-single-char*`

In a multi-line single quoted scalar, line breaks are subject to (flow) line folding, and any trailing white space is excluded from the content.

[148] `s-l-single-break(n) ::= s-ignored-white* b-l-folded-any(n,single)`

Example 4.59. Single Quoted Line Breaks

```
 'as space→↓
 trimmed·↓
↓
 specific⇓
↓
 none'
```

```
%YAML 1.1
---
!!str "as space \
  trimmed\n\
  specific\L\n\
  none"
```

```
Legend:
  s-l-single-break(n)
  s-ignored-white  s-white (Content)
```

A multi-line single quoted scalar consists of a (possibly empty) first line, any number of inner lines, and a final (possibly empty) last line.

```
[149] nb-single-multi(n) ::= nb-l-single-first(n)
                             l-single-inner(n)*
                             s-nb-single-last(n)
```

Leading white space in the first line is considered content only if followed by a non-space character.

```
[150] nb-l-single-first(n) ::= ( nb-single-char* ns-single-char )?
                               s-l-single-break(n)
```

Example 4.60. First Single Quoted Line

```
- '↓
  last'
- '·→↓
  last'
- '·→first↓
  last'
```

```
%YAML 1.1
---
!!seq [
  !!str " last",
  !!str " last",
  !!str " \tfirst last",
]
```

```
Legend:
  nb-l-single-first(n) s-ignored-white
```

All leading and trailing white space of inner lines is excludced from the content. Note that while prefix white space may contain tab characters, line indentation is restricted to space characters only. Unlike double quoted scalars, it is impossible to force the inclusion of the leading or trailing spaces in the content. Therefore, single quoted scalars lines can only be broken where a single space character separates two non-space characters.

```
[151] l-single-inner(n) ::= s-ignored-prefix(n,single) ns-single-char
                            ( nb-single-char* ns-single-char )?
                            s-l-single-break(n)
```

Example 4.61. Inner Single Quoted Lines

```
 'first
·→inner→↓
 last'
```

```
%YAML 1.1
---
!!str "first \
  inner \
  last"
```

```
Legend:
  l-single-inner(n)
  s-ignored-prefix(n,s) s-l-single-break(n)
```

The leading prefix white space of the last line is stripped in the same way as for inner lines. Trailing white space is considered content only if preceded by a non-space character.

```
[152] s-nb-single-last(n) ::= s-ignored-prefix(n,single)
                              ( ns-single-char nb-single-char* )?
```

Example 4.62. Last Single Quoted Lines

```
- 'first
··→'
- 'first

··→last'
```

```
%YAML 1.1
---
!!seq [
  !!str "first ",
  !!str "first\nlast",
]
```

```
Legend:
  s-nb-double-last(n) s-ignored-prefix(n,s)
```

4.5.1.3. Plain

The *plain style* uses no identifying indicators, and is therefore the most most limited and most context sensitive scalar style. Plain scalars can never contain any tab characters. They also must not contain the "`: `" and "` #`" character sequences as these combinations cause ambiguity with key: value pairs and comments. Inside flow collections, plain scalars are further restricted to avoid containing the "**[**", "**]**", "**{**", "**}**" and "**,**" characters as these would cause ambiguity with the flow collection structure (hence the need for the *flow-in context* and the *flow-out context*).

```
[153] nb-plain-char(c) ::= c = flow-out ⇒ nb-plain-char-out
                           c = flow-in  ⇒ nb-plain-char-in
                           c = flow-key ⇒ nb-plain-char-in
[154] nb-plain-char-out ::=   ( nb-char - “:” - “#” - #x9 /*TAB*/ )
                            | ( ns-plain-char(flow-out) “#” )
                            | ( “:” ns-plain-char(flow-out) )
[155] nb-plain-char-in ::= nb-plain-char-out - “,” - “[” - “]” - “{” - “}”
[156] ns-plain-char(c) ::= nb-plain-char(c) - #x20 /*SP*/
```

The first plain character is further restricted to avoid most indicators as these would cause ambiguity with various YAML structures. However, the first character may be "**-**", "**?**" or "**:**" provided it is followed by a non-space character.

```
[157] ns-plain-first-char(c) ::=   ( ns-plain-char(c) - c-indicator )
                                 | ( ( “-” | “?” | “:” ) ns-plain-char(c) )
```

Example 4.63. Plain Characters

```
# Outside flow collection:
- ::std::vector
- Up, up and away!
- -123
# Inside flow collection:
- [ ::std::vector,
  "Up, up and away!",
  -123 ]
```

```
Legend:
  ns-plain-first-char(c)
  ns-plain-char(c) Not ns-plain-char(c)
```

```
%YAML 1.1
---
!!seq [
  !!str "::std::vector",
  !!str "Up, up and away!",
  !!int "-123",
  !!seq [
    !!str "::std::vector",
    !!str "Up, up and away!",
    !!int "-123",
  ]
]
```

Plain scalars are restricted to a single line when contained inside a simple key.

```
[158] ns-plain(n,c) ::= c = flow-out ⇒ ns-plain-multi(n,c)?
                        c = flow-in  ⇒ ns-plain-multi(n,c)?
                        c = flow-key ⇒ ns-plain-single(c)
```

Example 4.64. Plain Scalars

```
simple key : {
  also simple : value,
  ? not a
  simple key : any
  value
}
```

```
Legend:
  ns-plain-single(c) ns-plain-multi(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "simple key"
  : !!map {
    ? !!str "also simple"
    : !!str "value",
    ? !!str "not a simple key"
    : !!str "any value"
  }
}
```

The first line of any flow scalar is indented according to the collection it is contained in. Therefore, there are two cases where a plain scalar begins on the first column of a line, without any preceding indentation spaces: a plain scalar used as a simple key of a non-indented block mapping, and any plain scalar nested in a non-indented flow collection. In these cases, the first line of the plain scalar must not conflict with a document boundary marker.

```
[159] l-forbidden-content ::= /* start of line */
                              ( c-document-start | c-document-end )
                              /* space or end of line */
```

Example 4.65. Forbidden Non-Indented Plain Scalar Content

```
---
---·||| : foo
...·>>>: bar
---
[
---↓
,
...·,
{
---·:
...·# Nested
}
]
...
```

```
ERROR:
 The --- and ... document
 start and end markers must
 not be specified as the
 first content line of a
 non-indented plain scalar.
```

YAML provides several easy ways to present such content without conflicting with the document boundary markers. For example:

Example 4.66. Document Marker Scalar Content

```
---
"---" : foo
...: bar
---
[
---,
...,
{
? ---
: ...
}
]
...
```

```
Legend:
  Content --- and ...
  Document marker --- and ...
```

```
%YAML 1.1
---
!!map {
  ? !!str "---"
  : !!str "foo",
  ? !!str "...",
  : !!str "bar"
}
%YAML 1.1
---
!!seq [
  !!str "---",
  !!str "...",
  !!map {
    ? !!str "---"
    : !!str "..."
  }
]
```

Thus, a single line plain scalar is a sequence of valid plain non-break printable characters, beginning and ending with non-space character and not conflicting with a document boundary markers. All characters are considered content, including any inner space characters.

```
[160] ns-plain-single(c) ::=   ( ns-plain-first-char(c)
                                 ( nb-plain-char(c)* ns-plain-char(c) )? )
                             - l-forbidden-content
```

In a multi-line plain scalar, line breaks are subject to (flow) line folding. Any prefix and trailing spaces are excluded from the content. Like single quoted scalars, in plain scalars it is impossible to force the inclusion of the leading or trailing spaces in the content. Therefore, plain scalars lines can only be broken where a single space character separates two non-space characters.

```
[161] s-l-plain-break(n) ::= s-ignored-white* b-l-folded-any(n,plain)
```

Example 4.67. Plain Line Breaks

```
 as space→↓
 trimmed·↓
↓
 specific⇓
↓
 none
```

```
%YAML 1.1
---
!!str "as space \
  trimmed\n\
  specific\L\n\
  none"
```

```
Legend:
  s-l-plain-break(n)
  s-ignored-white
```

A multi-line plain scalar contains additional continuation lines following the first line.

```
[162] ns-plain-multi(n,c) ::= ns-plain-single(c) s-ns-plain-more(n,c)*
```

Each continuation line must contain at least one non-space character. Note that it may be preceded by any number of empty lines.

```
[163] s-ns-plain-more(n,c) ::= s-l-plain-break(n)
                               s-ignored-prefix(n,plain) ns-plain-char(c)
                               ( nb-plain-char(c)* ns-plain-char(c) )?
```

Example 4.68. Plain Scalars

```
 first line·↓
···↓
··more line
```

```
%YAML 1.1
---
!!str "first line\n\
      more line"
```

```
Legend:
  ns-plain-single(c) s-l-plain-break(n)
  s-ignored-prefix(n,s) s-ns-plain-more(n,c)
```

4.5.2. Block Scalar Header

Block scalars are specified by several indicators given in a *header* preceding the content itself. The header is followed by an ignored line break (with an optional comment).

```
[164] c-b-block-header(s,m,t) ::= c-style-indicator(s)
                                  ( ( c-indentation-indicator(m)
                                      c-chomping-indicator(t) )
                                  | ( c-chomping-indicator(t)
                                      c-indentation-indicator(m) ) )
                                  s-b-comment
```

Example 4.69. Block Scalar Header

```
- | # Just the style↓
 literal
- >1 # Indentation indicator↓
 ·folded
- |+ # Chomping indicator↓
 keep

- >-1 # Both indicators↓
 ·strip
```

```
%YAML 1.1
---
!!seq [
  !!str "literal\n",
  !!str "·folded\n",
  !!str "keep\n\n",
  !!str "·strip",
]
```

```
Legend:
  c-b-block-header(s,m,t)
```

4.5.2.1. Block Style Indicator

The first character of the block scalar header is either “**|**” for a literal scalar or “**>**” for a folded scalar.

```
[165] c-style-indicator(s) ::= s = literal ⇒ “|”
                               s = folded  ⇒ “>”
```

Example 4.70. Block Style Indicator

```
- |
 literal
- >
 folded
```

```
Legend:
  c-style-indicator(s)
```

```
%YAML 1.1
---
!!seq [
  !!str "literal\n",
  !!str "folded\n",
]
```

4.5.2.2. Block Indentation Indicator

Typically, the indentation level of a block scalar is detected from its first non-empty line. This detection fails when this line contains leading space characters (note it may safely start with a tab or a “**#**” character). When detection fails, YAML requires that the indentation level for the content be given using an explicit *indentation indicator*. This level is specified as the integer number of the additional indentation spaces used for the content. If the block scalar begins with leading empty lines followed by a non-empty line, the indentation level is deduced from the non-empty line. In this case, it is an error for any such leading empty line to contain more spaces than the indentation level deduced from the non-empty line. It is always valid to specify an indentation indicator for a block scalar node, though a YAML processor should only do so in cases where detection will fail.

```
[166] c-indentation-indicator(m) ::= explicit(m) ⇒ ns-dec-digit - “0”
                                     detect(m)   ⇒ /* empty */
```

Example 4.71. Block Indentation Indicator

```
- |
·detected
- >
·
··
··# detected
- |1
··explicit
- >
·→
·detected
```

```
%YAML 1.1
---
!!seq [
  !!str "detected\n",
  !!str "\n\n# detected\n",
  !!str "·explicit\n",
  !!str "\t·detected\n",
]
```

```
Legend:
  c-indentation-indicator(m)
  s-indent(n)
```

Example 4.72. Invalid Block Scalar Indentation Indicators

```
- |
··
·text
- >
··text
·text
- |1
·text
```

```
ERROR:
- A leading all-space line must
  not have too many spaces.
- A following text line must
  not be less indented.
- The text is less indented
  than the indicated level.
```

4.5.2.3. Block Chomping Indicator

YAML supports three possible block *chomping* methods:

Strip
: *Stripping* is specified using the “**-**” *chomping indicator*. In this case, the line break character of the last non-empty line (if any) is excluded from the scalar's content. Any trailing empty lines are considered to be (empty) comment lines and are also discarded.

Clip
: *Clipping* the default behavior used if no explicit chomping indicator is specified. In this case, The line break character of the last non-empty line (if any) is preserved in the scalar's content. However, any trailing empty lines are considered to be (empty) comment lines and are discarded.

Keep
: *Keeping* is specified using the “**+**” *chomping indicator*. In this case, the line break character of the last non-empty line (if any) is preserved in the scalar's content. In addition, any trailing empty lines are each considered to present a single trailng content line break. Note that these line breaks are not subject to folding.

The chomping method ised is a presentation detail and is not reflected in the serialization tree (and hence the representation graph).

```
[167] c-chomping-indicator(t) ::= t = strip ⇒ “-”
                                  t = clip  ⇒ /* empty */
                                  t = keep  ⇒ “+”
```

Thus, the final line break of a block scalar may be included or excluded from the content, depending on the specified chomping indicator.

```
[168] b-chomped-last(t) ::= t = strip ⇒ b-strip-last
                            t = clip  ⇒ b-keep-last
                            t = keep  ⇒ b-keep-last
[169] b-strip-last ::= b-ignored-any
[170] b-keep-last ::= b-normalized
```

Example 4.73. Chomping Final Line Break

```
strip: |-
  text¶
clip: |
  text↓
keep: |+
  text⇓
```

```
Legend:
  b-strip-last
  b-keep-last
```

```
%YAML 1.1
---
!!map {
  ? !!str "strip"
  : !!str "text",
  ? !!str "clip"
  : !!str "text\n",
  ? !!str "keep"
  : !!str "text\L",
}
```

Similarly, empty lines immediately following the block scalar may be interpreted either as presenting trailing line breaks or as (empty) comment lines, depending on the specified chomping indicator.

```
[171] l-chomped-empty(n,t) ::= t = strip ⇒ l-strip-empty(n)
                               t = clip  ⇒ l-strip-empty(n)
                               t = keep  ⇒ l-keep-empty(n)
[172] l-strip-empty(n) ::= ( s-indent(≤n) b-ignored-any )* l-trail-comments(n)?
[173] l-keep-empty(n) ::= l-empty(n,literal)* l-trail-comments(n)?
```

Explicit comment lines may then follow. To prevent ambiguity, the first such comment line must be less indented than the block scalar content. Additional comment lines, if any, are not so restricted.

```
[174] l-trail-comments(n) ::= s-indent(<n) c-nb-comment-text b-ignored-any
                              l-comment*
```

Example 4.74. Block Scalar Chomping

```
 # Strip
  # Comments:
strip: |-
  # text¶
··⇓
·# Clip
··# comments:
↓
clip: |
  # text↓
·¶
·# Keep
··# comments:
↓
keep: |+
  # text⇓
↓
·# Trail
··# comments.
```

```
%YAML 1.1
---
!!seq [
  ? !!str "strip"
  : !!str "text",
  ? !!str "clip"
  : !!str "text\n",
  ? !!str "keep"
  : !!str "text\L\n",
]
```

```
Legend:
  l-strip-empty(n)
  l-keep-empty(n)
  l-trail-comments(n)
```

Note that if a block scalar consists of only empty lines, then these lines are considered trailing lines and hence are affected by chomping.

Example 4.75. Empty Scalar Chomping

```
strip: >-
↓
clip: >
↓
keep: |+
↓
```

```
Legend:
  l-strip-empty(n)
  l-keep-empty(n)
```

```
%YAML 1.1
---
!!seq [
  ? !!str "strip"
  : !!str "",
  ? !!str "clip"
  : !!str "",
  ? !!str "keep"
  : !!str "\n",
]
```

4.5.3. Block Scalar Styles

YAML provides two *Block scalar styles*, literal and folded. The block scalar content is is ended by a less-indented line or the end of the characters stream.

4.5.3.1. Literal

The *literal style* is the simplest, most restricted and most readable scalar style. It is especially suitable for source code or other text containing significant use of indicators, escape sequences and line breaks. In particular, literal content lines may begin with a tab or a “**#**” character.

```
[175] c-l+literal(n) ::= c-b-block-header(literal,m,t)
                         l-literal-content(n+m,t)
```

Example 4.76. Literal Scalar

```
| # Simple block scalar↓
 literal↓
 →text↓
```

```
Legend:
  c-b-block-header(s,m,t)
  l-literal-content(n,t)
```

```
%YAML 1.1
---
!!seq [
  !!str "literal\n\
        \ttext\n"
]
```

Inside literal scalars, each non-empty line may be preceded by any number of empty lines. No processing is performed on these lines except for stripping the indentation. In particular, such lines are never folded. Literal non-empty lines may include only spaces, tabs, and other printable characters.

```
[176] l-nb-literal-text(n) ::= l-empty(n,block)* s-indent(n) nb-char+
```

The line break following a non-empty inner literal line is normalized. Again, such line breaks are never folded.

```
[177] l-literal-inner(n) ::= l-nb-literal-text(n) b-normalized
```

Example 4.77. Inner Literal Lines

```
|
·
··
··literal↓
·
··text↓
↓
·# Comment
```

```
%YAML 1.1
---
!!str "\nliteral\n\ntext\n"
```

```
Legend:
  l-nb-literal-text(n)
  l-nb-literal-inner(n)
```

The line break following the final non-empty literal line is subject to chomping.

```
[178] l-literal-last(n,t) ::= l-nb-literal-text(n) b-chomped-last(t)
```

Trailing empty lines following the last literal non-empty line, if any, are also subject to chomping.

```
[179] l-literal-content(n,t) ::= ( l-literal-inner(n)* l-literal-last(n,t) )?
                                 l-chomped-empty(n,t)?
```

Example 4.78. Last Literal Line

```
|
·
··
··literal↓
·
··text↓
↓
·# Comment
```

```
%YAML 1.1
---
!!str "\nliteral\n\ntext\n"
```

```
Legend:
  l-nb-literal-text(n)
  l-nb-literal-last(n,t)
  b-chomped-last(t)
  l-chomped-empty(n,t)
```

4.5.3.2. Folded

The *folded style* is similar to the literal style. However, unlike literal content, folded content is subject to (block) line folding.

```
[180] c-l+folded(n) ::= c-b-block-header(folded,m,t)
                        l-folded-content(n+m,t)
```

Example 4.79. Folded Scalar

```
> # Simple folded scalar↓
 folded↓
text↓
→lines↓
```

```
%YAML 1.1
---
!!seq [
  !!str "folded text\n\
        \ttext\n"
]
```

```
Legend:
  c-b-block-header(s,m,t)
  l-folded-content(n,t)
```

Line folding allows long content lines to be broken anywhere a single space character separates two non-space characters.

```
[181] s-nb-folded-line(n) ::= s-indent(n) ns-char nb-char*
[182] l-nb-folded-lines(n) ::= ( s-nb-folded-line(n)
                                 b-l-folded-any(n,folded) )*
                               s-nb-folded-line(n)
```

Example 4.80. Folded Lines

```
>
·folded↓
·line↓
↓
·next
·line↓

   * bullet
   * list

·last↓
·line↓

# Comment
```

```
%YAML 1.1
---
!!seq [
  !!str "folded line\n\
        next line\n\
        \  * bullet\n\
        \  * list\n\
        last line\n"
]
```

```
Legend:
  l-nb-folded-lines(n)
```

Lines starting with white space characters (*"more indented" lines*) are not folded. Note that folded scalars, like literal scalars, may contain tab characters. However, any such characters must be properly indented using only space characters.

```
[183] b-l-spaced(n) ::= b-normalized l-empty(n,folded)*
[184] s-nb-spaced-text(n) ::= s-indent(n) s-white nb-char*
[185] l-nb-spaced-lines(n) ::= ( s-nb-spaced-text(n) b-l-spaced(n) )*
                                 s-nb-spaced-text(n)
```

Example 4.81. Spaced Lines

```
>
 folded
 line

 next
 line

···* bullet↓
···* list↓

 last
 line

# Comment
```

```
%YAML 1.1
---
!!seq [
  !!str "folded line\n\
        next line\n\
        \  * bullet\n\
        \  * list\n\
        last line\n"
]
```

```
Legend:
  l-nb-spaced-lines(n)
```

Folded content may start with either line type. If the content begins with a “more indented” line (starting with spaces), an indentation indicator must be specified in the block header. Note that leading empty lines and empty lines separating lines of a different type are never folded.

```
[186] l-nb-start-with-folded(n) ::= l-empty(n,block)* l-nb-folded-lines(n)
                                    ( b-normalized l-nb-start-with-spaced(n) )?
[187] l-nb-start-with-spaced(n) ::= l-empty(n,block)* l-nb-spaced-lines(n)
                                    ( b-normalized l-nb-start-with-folded(n) )?
[188] l-nb-start-with-any(n) ::=   l-nb-start-with-folded(n)
                                 | l-nb-start-with-spaced(n)
```

Example 4.82. Empty Separation Lines

```
>
 folded
 line

 next
 line↓
↓
   * bullet
   * list↓
↓
 last
 line

# Comment
```

```
%YAML 1.1
---
!!seq [
  !!str "folded line\n\
        next line\n\
        \  * bullet\n\
        \  * list\n\
        last line\n"
]
```

Legend:
  b-normalized l-empty(n,s)

The final line break, and trailing empty lines, if any, are subject to chomping and are never folded.

```
[189] l-folded-content(n,t) ::= ( l-nb-start-with-any(n) b-chomped-last(t) )?
                                l-chomped-empty(n,t)
```

Example 4.83. Final Empty Lines

```
>
 folded
 line

 next
 line

   * bullet
   * list

 last
 line↓
↓
# Comment
```

```
%YAML 1.1
---
!!seq [
  !!str "folded line\n\
        next line\n\
        \  * bullet\n\
        \  * list\n\
        last line\n"
]
```

Legend:
  b-chomped-last(t) l-chomped-empty(n,t)

4.6. Collection Styles

Collection content can be presented in a single *flow style* and a single *block style* for each of the two collection kinds (sequence and mapping). In addition, YAML provides several in-line compact syntax forms for improved readability of common special cases. In all cases, the collection style is a presentation detail and must not be used to convey content information.

A flow collection may be nested within a block collection (flow-out context), nested within another flow collection (flow-in context), or be a part of a simple key (flow-key context). Flow collection entries are separated by the "**,**" *indicator*. The final "**,**" may be ommitted. This does not cause ambiguity because flow collection entries can never be completely empty.

```
[190] in-flow(c) ::= c = flow-out ⇒ flow-in
                     c = flow-in  ⇒ flow-in
                     c = flow-key ⇒ flow-key
```

4.6.1. Sequence Styles

Sequence content is an ordered collection of sub-nodes. Comments may be interleaved between the sub-nodes. Sequences may be presented in a flow style or a block style. YAML provides compact notations for in-line nesting of a collection in a block sequence and for nesting a single pair mapping in a flow sequence.

4.6.1.1. Flow Sequences

Flow sequence content is denoted by surrounding "***[***" and "***]***" characters.

```
[191] c-flow-sequence(n,c) ::= "[" s-separate(n,c)?
                               ns-s-flow-seq-inner(n,c)*
                               ns-s-flow-seq-last(n,c)?
                               "]"
```

Sequence entries are separated by a "**,**" character.

```
[192] ns-s-flow-seq-inner(n,c) ::= ns-s-flow-seq-entry(n,c) "," s-separate(n,c)?
```

The final entry may omit the "**,**" character. This does not cause ambiguity since sequence entries must not be completely empty.

```
[193] ns-s-flow-seq-last(n,c) ::= ns-s-flow-seq-entry(n,c)
```

Example 4.84. Flow Sequence

```
- [ inner, inner, ]
- [inner,last]
```

```
Legend:
  c-sequence-start c-sequence-end
  ns-s-flow-seq-inner(n,c)
  ns-s-flow-seq-last(n,c)
```

```
%YAML 1.1
---
!!seq [
  !!seq [
    !!str "inner",
    !!str "inner",
  ],
  !!seq [
    !!str "inner",
    !!str "last",
  ],
]
```

Any flow node may be used as a flow sequence entry. In addition, YAML provides a compact form for the case where a flow sequence entry is a mapping with a single key: value pair, and neither the mapping node nor its single key node have any properties specified.

```
[194] ns-s-flow-seq-entry(n,c) ::=   ( ns-flow-node(n,in-flow(c))
                                        s-separate(n,in-flow(c))? )
                                    | ns-s-flow-single-pair(n,in-flow(c))
```

Example 4.85. Flow Sequence Entries

```
[
"double
 quoted", 'single
                quoted',
plain
 text", [ nested ],
single: pair ,
]
```

```
Legend:
  ns-flow-node(n,c)
  ns-s-flow-single-pair(n,c)
```

```
%YAML 1.1
---
!!seq [
  !!str "double quoted",
  !!str "single quoted",
  !!str "plain text",
  !!seq [
    !!str "nested",
  ],
  !!map {
    ? !!str "single"
    : !!str "pair"
  }
]
```

4.6.1.2. Block Sequences

A *block sequence* is simply a series of entries, each presenting a single node.

```
[195] c-l-block-sequence(n,c) ::= c-l-comments l-block-seq-entry(n,c)+
```

Example 4.86. Block Sequence

```
block: # Block
       # sequence↓
- one↓
- two : three↓
```

```
Legend:
  c-l-comments
  l-block-seq-entry(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "block"
  : !!seq [
    !!str "one",
    !!str "two"
  ]
}
```

Each block sequence entry is denoted by a leading "**-**" *indicator*, separated by spaces from the entry node.

```
[196] l-block-seq-entry(n,c) ::= s-indent(seq-spaces(n,c)) "-"
                                 s-l+block-indented(seq-spaces(n,c),c)
```

People read the "**-**" character as part of the indentation. Hence, block sequence entries require one less space of indentation, unless the block sequence is nested within another block sequence (hence the need for the *block-in context* and *block-out context*).

```
[197] seq-spaces(n,c) ::= c = block-out ⇒ n-1
                          c = block-in  ⇒ n
```

Example 4.87. Block Sequence Entry Indentation

```
block:
- one
-
·- two
```

```
Legend:
  s-indent(n)
  s-l+block-indented(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "block"
  : !!seq [
    !!str "one",
    !!seq [
      !!str "two"
    ]
  ]
}
```

The entry node may be either completely empty, a normal block node, or use a compact in-line form.

```
[198] s-l+block-indented(n,c) ::=   s-l-empty-block
                                  | s-l+block-node(n,c)
                                  | s-l+block-in-line(n)
```

The compact *in-line* form may be used in the common case when the block sequence entry is itself a block collection, and neither the collection entry nor its first nested node have any properties specified. In this case, the nested collection may be specified in the same line as the “**-**” character, and any following spaces are considered part of the in-line nested collection's indentation.

```
[199] s-l+block-in-line(n) ::= s-indent(m>0)
                               ( ns-l-in-line-sequence(n+1+m)
                               | ns-l-in-line-mapping(n+1+m) )
```

An *in-line block sequence* begins with an indented same-line sequence entry, followed by optional additional normal block sequence entries, properly indented.

```
[200] ns-l-in-line-sequence(n) ::= “-” s-l+block-indented(n,block-out)
                                   l-block-seq-entry(n,block-out)*
```

Example 4.88. Block Sequence Entry Types

```
-  # Empty
-  |
 block node
-  - one # in-line
   - two # sequence
-  one: two # in-line
                # mapping
```

```
Legend:
  s-l-empty-block
  s-l+block-node(n,c)
  s-l+block-in-line(n)
```

```
%YAML 1.1
---
!!seq [
  !!str "",
  !!str "block node\n",
  !!seq [
    !!str "one",
    !!str "two",
  ]
  !!map {
    ? !!str "one"
    : !!str "two",
  }
]
```

4.6.2. Mapping Styles

A *mapping node* is an unordered collection of *key: value* pairs. Of necessity, these pairs are presented in some order in the characters stream. As a serialization detail, this key order is preserved in the serialization tree. However it is not reflected in the representation graph and hence must not be used when constructing native data structures. It is an error for two equal keys to appear in the same mapping value. In such a case the YAML processor may continue, ignoring the second key: value pair and issuing an appropriate warning. This strategy preserves a consistent information model for one-pass and random access applications.

4.6.2.1. Flow Mappings

Flow mapping content is denoted by surrounding “**{**” and “**}**” characters.

```
[201] c-flow-mapping(n,c) ::= “{” s-separate(n,c)?
                              ns-s-flow-map-inner(n,c)*
                              ns-s-flow-map-last(n,c)?
                              “}”
```

Mapping entries are separated by a “`,`” character.

[202] `ns-s-flow-map-inner(n,c) ::= ns-s-flow-map-entry(n,c) “,” s-separate(n,c)?`

The final entry may omit the “`,`” character. This does not cause ambiguity since mapping entries must not be completely empty.

[203] `ns-s-flow-map-last(n,c) ::= ns-s-flow-map-entry(n,c)`

Example 4.89. Flow Mappings

```
- { inner : entry , also: inner , }
- {inner: entry, last : entry}
```

```
Legend:
  c-mapping-start c-mapping-end
  ns-s-flow-map-inner(n,c)
  ns-s-flow-map-last(n,c)
```

```
%YAML 1.1
---
!!seq [
  !!map {
    ? !!str "inner"
    : !!str "entry",
    ? !!str "also"
    : !!str "inner"
  },
  !!map {
    ? !!str "inner"
    : !!str "entry",
    ? !!str "last"
    : !!str "entry"
  }
]
```

Flow mappings allow two forms of keys: explicit and simple.

Explicit Keys An *explicit key* is denoted by the *“**?**” indicator*, followed by separation spaces.

```
[204] s-flow-separated(n,c) ::=   ( s-separate(n,c) ns-flow-node(n,in-flow(c))
                                     s-separate(n,c)? )
                                 | ( e-empty-flow s-separate(n,c) )
[205] c-s-flow-explicit-key(n,c) ::= “?” s-flow-separated(n,c)
```

Simple Keys A *simple key* has no identifying mark. It is recognized as being a key either due to being inside a flow mapping, or by being followed by an explicit value. Hence, to avoid unbound lookahead in YAML processors, simple keys are restricted to a single line and must not span more than 1024 stream characters (hence the need for the *flow-key context*). Note the 1024 character limit is in terms of Unicode characters rather than stream octets, and that it includes the separation following the key itself.

[206] `ns-s-flow-simple-key(n,c) ::= ns-flow-node(n,flow-key) s-flow-separated(n,c)?`

Example 4.90. Flow Mapping Keys

```
{
? : value # Empty key
? explicit
 key: value,
simple key : value
[ collection, simple, key ]: value
}
```

```
Legend:
  c-s-flow-explicit-key(n,c)
  ns-s-flow-simple-key(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str ""
  : !!str "value",
  ? !!str "explicit key"
  : !!str "value",
  ? !!str "simple key"
  : !!str "value",
  ? !!seq [
    !!str "collection",
    !!str "simple",
    !!str "key"
  ]
  : !!str "value"
}
```

Example 4.91. Invalid Flow Mapping Keys

```
{
multi-line
 simple key : value,
very long ...(>1KB)... key: value
}
```

```
ERROR:
- A simple key is restricted
  to only one line.
- A simple key msut not be
  longer than 1024 bytes.
```

Flow mappings also allow two forms of values, explicit and completely empty.

Explicit Values An *explicit value* is denoted by the *“**:**” indicator*, followed by separation spaces.

[207] `c-s-flow-explicit-value(n,c) ::= “:” s-flow-separated(n,c)`

Example 4.92. Flow Mapping Values

```
{
key : value,
empty:° # empty value↓
}
```

```
Legend:
  c-s-flow-explicit-value(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "key"
  : !!str "value",
  ? !!str "empty"
  : !!str "",
}
```

Thus, there are four possible combinations for a flow mapping entry:

- Explicit key and explicit value:

```
[208] c-s-flow-explicit-explicit(n,c) ::= c-s-flow-explicit-key(n,c)
                                          c-s-flow-explicit-value(n,c)
```

- Explicit key and completely empty value:

```
[209] c-s-flow-explicit-empty(n,c) ::= c-s-flow-explicit-key(n,c) e-empty-flow
```

- Simple key and explicit value:

```
[210] ns-s-flow-simple-explicit(n,c) ::= ns-s-flow-simple-key(n,c)
                                         c-s-flow-explicit-value(n,c)
```

- Simple key and completely empty value:

```
[211] ns-s-flow-simple-empty(n,c) ::= ns-s-flow-simple-key(n,c) e-empty-flow
```

Inside flow mappings, all four combinations may be used.

```
[212] ns-s-flow-map-entry(n,c) ::=   c-s-flow-explicit-explicit(n,c)
                                   | c-s-flow-explicit-empty(n,c)
                                   | ns-s-flow-simple-explicit(n,c)
                                   | ns-s-flow-simple-empty(n,c)
```

Example 4.93. Flow Mapping Key: Value Pairs

```
{
? explicit key1 : Explicit value,
? explicit key2 :° , # Explicit empty
? explicit key3,     # Empty value
simple key1 : explicit value,
simple key2 :° ,     # Explicit empty
simple key3,         # Empty value
}
```

```
Legend:
  c-s-flow-explicit-explicit(n,c)
  c-s-flow-explicit-empty(n,c)
  ns-s-flow-simple-explicit(n,c)
  ns-s-flow-simple-empty(n,c)
```

```
%YAML 1.1
---
!!map {
  ? !!str "explicit key1"
  : !!str "explicit value",
  ? !!str "explicit key2"
  : !!str "",
  ? !!str "explicit key3"
  : !!str "",
  ? !!str "simple key1"
  : !!str "explicit value",
  ? !!str "simple key2"
  : !!str "",
  ? !!str "simple key3"
  : !!str "",
}
```

YAML also allows omitting the surrounding "**{**" and "**}**" characters when nesting a flow mapping in a flow sequence if the mapping consists of a *single key: value pair* and neither the mapping nor the key have any properties specified. In this case, only three of the combinations may be used, to prevent ambiguity.

```
[213] ns-s-flow-single-pair(n,c) ::=   c-s-flow-explicit-explicit(n,c)
                                     | c-s-flow-explicit-empty(n,c)
                                     | ns-s-flow-simple-explicit(n,c)
```

Example 4.94. Single Pair Mappings

```
[
? explicit key1 : explicit value,
? explicit key2 :° , # Explicit value
? explicit key3,     # Empty value
simple key1 : explicit value,
simple key2 :° ,     # Explicit empty
]
```

```
Legend:
  c-s-flow-explicit-explicit(n,c)
  c-s-flow-explicit-empty(n,c)
  ns-s-flow-simple-explicit(n,c)
```

```
%YAML 1.1
---
!!seq [
  !!map {
    ? !!str "explicit key1"
    : !!str "explicit value",
  },
  !!map {
    ? !!str "explicit key2"
    : !!str "",
  },
  !!map {
    ? !!str "explicit key3"
    : !!str "",
  },
  !!map {
    ? !!str "simple key1"
    : !!str "explicit value",
  },
]
```

4.6.2.2. Block Mappings

A *Block mapping* is simply a series of entries, each presenting a key: value pair.

```
[214] c-l-block-mapping(n) ::= c-l-comments
                               ( s-indent(n) ns-l-block-map-entry(n) )+
```

Example 4.95. Block Mappings

```
block: # Block
    # mapping↓
·key: value↓
```

```
Legend:
  c-l-comments
  s-indent(n)
  ns-l-block-map-entry(n)
```

```
%YAML 1.1
---
!!map {
  ? !!str "block"
  : !!map {
    !!str "key",
    !!str "value"
  }
}
```

A block mapping entry may be presented using either an explicit or a simple key.

```
[215] ns-l-block-map-entry(n) ::=   ns-l-block-explicit-entry(n)
                                  | ns-l-block-simple-entry(n)
```

Explicit Key Entries

Explicit key nodes are denoted by the “**?**” character. YAML allows here the same inline compact notation described above for block sequence entries, in which case the “**?**” character is considered part of the key's indentation.

```
[216] ns-l-block-explicit-key(n) ::= “?” s-l+block-indented(n,block-out)
```

- In an explicit key entry, value nodes begin on a separate line and are denoted by by the “**:**” character. Here again YAML allows the use of the inline compact notation which case the “**:**” character is considered part of the values's indentation.

```
[217] l-block-explicit-value(n) ::= s-indent(n) “:”
                                    s-l+block-indented(n,block-out)
```

- An explicit key entry may also use a completely empty value.

```
[218] ns-l-block-explicit-entry(n) ::= ns-l-block-explicit-key(n)
                                       ( l-block-explicit-value(n)
                                       | e-empty-flow )
```

Example 4.96. Explicit Block Mapping Entries

```
? explicit key # implicit value↓°
? |
  block key↓
: - one # explicit in-line
  - two # block value↓
```

```
Legend:
  ns-l-block-explicit-key(n)
  l-block-explicit-value(n)
  e-empty-flow
```

```
%YAML 1.1
---
!!map {
  ? !!str "explicit key"
  : !!str "",
  ? !!str "block key\n"
  : !!seq [
    !!str "one",
    !!str "two",
  ]
}
```

Simple Key Entries

YAML allows the “**?**” character to be omitted for simple keys. Similarly to flow mapping, such a key is recognized by a following “**:**” character. Again, to avoid unbound lookahead in YAML processors, simple keys are restricted to a single line and must not span more than 1024 stream characters. Again, this limit is in terms of Unicode characters rather than stream octets, and includes the separation following the key, if any.

```
[219] ns-block-simple-key(n) ::= ns-flow-node(n,flow-key)
                                 s-separate(n,block-out)? “:”
```

- In a simple key entry, an explicit value node may be presented in the same line. Note however that in this case, the key is not considered to be a form of indentation, hence the compact in-line notation must not be used. The value following the simple key may also be completely empty.

```
[220] s-l+block-simple-value(n) ::=   s-l+block-node(n,block-out)
                                    | s-l-empty-block
[221] ns-l-block-simple-entry(n) ::= ns-block-simple-key(n)
                                     s-l+block-simple-value(n)
```

Example 4.97. Simple Block Mapping Entries

```
plain key:° # empty value↓
"quoted key":↓
- one # explicit next-line
- two # block value↓
```

```
Legend:
  ns-block-simple-key(n)
  s-l+block-simple-value(n)
```

```
%YAML 1.1
---
!!map {
  ? !!str "plain key"
  : !!str "",
  ? !!str "quoted key\n"
  : !!seq [
    !!str "one",
    !!str "two",
  ]
}
```

An *in-line block mapping* begins with a same-line mapping entry, followed by optional additional normal block mapping entries, properly indented.

```
[222] ns-l-in-line-mapping(n) ::= ns-l-block-map-entry(n)
                                  ( s-indent(n) ns-l-block-map-entry(n) )*
```

Example 4.98. In-Line Block Mappings

```
- sun: yellow↓
- ? earth: blue↓
  : moon: white↓
```

```
Legend:
  ns-l-in-line-mapping(n)
```

```
%YAML 1.1
---
!!seq {
  !!map {
    ? !!str "sun"
    : !!str "yellow",
  },
  !!map {
    ? !!map {
      ? !!str "earth"
      : !!str "blue"
    }
    : !!map {
      ? !!str "moon"
      : !!str "white"
    }
  }
}
```

Terms Index