

YAML Ain't Markup Language (YAML)

Version 1.2

3rd Edition, Patched at 2009-10-01

Oren Ben-Kiki

<oren@ben-kiki.org>

Clark Evans

<cce@clarkevans.com>

Ingy döt Net

<ingy@ingy.net>

Latest (patched) version:

HTML: <http://yaml.org/spec/1.2/spec.html>

PDF: <http://yaml.org/spec/1.2/spec.pdf>

PS: <http://yaml.org/spec/1.2/spec.ps>

Errata: <http://yaml.org/spec/1.2/errata.html>

Previous (original) version: <http://yaml.org/spec/1.2/2009-07-21/spec.html>

Copyright © 2001-2009 Oren Ben-Kiki, Clark Evans, Ingy döt Net

This document may be freely copied, provided it is not modified.

Status of this Document

This document reflects the third version of YAML data serialization language. The content of the specification was arrived at by consensus of its authors and through user feedback on the [yaml-core](#) mailing list. We encourage implementers to please update their software with support for this version.

The primary objective of this revision is to bring YAML into compliance with JSON as an official subset. YAML 1.2 is compatible with 1.1 for most practical applications - this is a minor revision. An expected source of incompatibility with prior versions of YAML, especially the syck implementation, is the change in implicit typing rules. We have removed unique implicit typing rules and have updated these rules to align them with JSON's productions. In this version of

YAML, boolean values may be serialized as “**true**” or “**false**”; the empty scalar as “**null**”. Unquoted numeric values are a superset of JSON's numeric production. Other changes in the specification were the removal of the Unicode line breaks and production bug fixes. We also define 3 built-in implicit typing rule sets: untyped, strict JSON, and a more flexible YAML rule set that extends JSON typing.

The difference between late 1.0 drafts which syck 0.55 implements and the 1.1 revision of this specification is much more extensive. We fixed usability issues with the tagging syntax. In particular, the single exclamation was re-defined for private types and a simple prefixing mechanism was introduced. This revision also fixed many production edge cases and introduced a type repository. Therefore, there are several incompatibilities between syck and this revision as well.

The list of known errors in this specification is available at <http://yaml.org/spec/1.2/errata.html>. Please report errors in this document to the [yaml-core](#) mailing list. This revision contains fixes for all errors known as of 2009-10-01.

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 types 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](#), this specification provides all the information necessary to understand YAML Version 1.2 and to create programs that process YAML information.

Table of Contents

1. Introduction

1.1. Goals

1.2. Prior Art

1.3. Relation to JSON

1.4. Relation to XML

1.5. Terminology

[2. Preview](#)

[2.1. Collections](#)

[2.2. Structures](#)

[2.3. Scalars](#)

[2.4. Tags](#)

[2.5. Full Length Example](#)

[3. Processing YAML Information](#)

[3.1. Processes](#)

[3.1.1. Dump](#)

[3.1.2. Load](#)

[3.2. Information Models](#)

[3.2.1. Representation Graph](#)

[3.2.1.1. Nodes](#)

[3.2.1.2. Tags](#)

[3.2.1.3. Node Comparison](#)

[3.2.2. Serialization Tree](#)

[3.2.2.1. Keys Order](#)

[3.2.2.2. Anchors and Aliases](#)

[3.2.3. Presentation Stream](#)

[3.2.3.1. Node Styles](#)

[3.2.3.2. Scalar Formats](#)

[3.2.3.3. Comments](#)

[3.2.3.4. Directives](#)

[3.3. Loading Failure Points](#)

[3.3.1. Well-Formed Streams and Identified Aliases](#)

[3.3.2. Resolved Tags](#)

[3.3.3. Recognized and Valid Tags](#)

[3.3.4. Available Tags](#)

4. Syntax Conventions

4.1. Production Parameters

4.2. Production Naming Conventions

[5. Characters](#)

[5.1. Character Set](#)

[5.2. Character Encodings](#)

[5.3. Indicator Characters](#)

[5.4. Line Break Characters](#)

[5.5. White Space Characters](#)

[5.6. Miscellaneous Characters](#)

[5.7. Escaped Characters](#)

[6. Basic Structures](#)

[6.1. Indentation Spaces](#)

[6.2. Separation Spaces](#)

[6.3. Line Prefixes](#)

[6.4. Empty Lines](#)

[6.5. Line Folding](#)

[6.6. Comments](#)

[6.7. Separation Lines](#)

[6.8. Directives](#)

[6.8.1. “YAML” Directives](#)

[6.8.2. “TAG” Directives](#)

[6.8.2.1. Tag Handles](#)

[6.8.2.2. Tag Prefixes](#)

[6.9. Node Properties](#)

[6.9.1. Node Tags](#)

[6.9.2. Node Anchors](#)

[7. Flow Styles](#)

[7.1. Alias Nodes](#)

[7.2. Empty Nodes](#)

[7.3. Flow Scalar Styles](#)

[7.3.1. Double-Quoted Style](#)

[7.3.2. Single-Quoted Style](#)

[7.3.3. Plain Style](#)

[7.4. Flow Collection Styles](#)

[7.4.1. Flow Sequences](#)

[7.4.2. Flow Mappings](#)

[7.5. Flow Nodes](#)

[8. Block Styles](#)

[8.1. Block Scalar Styles](#)

[8.1.1. Block Scalar Headers](#)

[8.1.1.1. Block Indentation Indicator](#)

[8.1.1.2. Block Chomping Indicator](#)

[8.1.2. Literal Style](#)

[8.1.3. Folded Style](#)

[8.2. Block Collection Styles](#)

[8.2.1. Block Sequences](#)

[8.2.2. Block Mappings](#)

[8.2.3. Block Nodes](#)

[9. YAML Character Stream](#)

[9.1. Documents](#)

[9.1.1. Document Prefix](#)

[9.1.2. Document Markers](#)

[9.1.3. Bare Documents](#)

[9.1.4. Explicit Documents](#)

[9.1.5. Directives Documents](#)

[9.2. Streams](#)

[10. Recommended Schemas](#)

[10.1. Failsafe Schema](#)

[10.1.1. Tags](#)

[10.1.1.1. Generic Mapping](#)

[10.1.1.2. Generic Sequence](#)

[10.1.1.3. Generic String](#)

[10.1.2. Tag Resolution](#)

[10.2. JSON Schema](#)

[10.2.1. Tags](#)

[10.2.1.1. Null](#)

[10.2.1.2. Boolean](#)

[10.2.1.3. Integer](#)

[10.2.1.4. Floating Point](#)

[10.2.2. Tag Resolution](#)

[10.3. Core Schema](#)

[10.3.1. Tags](#)

[10.3.2. Tag Resolution](#)

[10.4. Other Schemas](#)

[Index](#)

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 specification 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 [key: value pairs](#), and [dashes](#) are used to create “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 [native data structure](#). While most programming languages can use YAML for data serialization, YAML excels in working with 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, in decreasing priority:

1. YAML is easily readable by humans.
2. YAML data is portable between programming languages.
3. YAML matches the [native data structures](#) of agile 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](#). Later on, it directly incorporated experience from Ingy döt Net's Perl module [Data::Denter](#). Since then, YAML has matured through ideas and support from its user community.

YAML integrates and builds upon concepts described by [C](#), [Java](#), [Perl](#), [Python](#), [Ruby](#), [RFC0822](#) (MAIL), [RFC1866](#) (HTML), [RFC2045](#) (MIME), [RFC2396](#) (URI), [XML](#), [SAX](#), [SOAP](#), and [JSON](#).

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 JSON's and 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 [scalar content](#).

YAML's core type system is based on the requirements of agile languages such as Perl, Python, and Ruby. YAML directly supports both [collections](#) ([mappings](#), [sequences](#)) and [scalars](#). Support for these 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](#) a graph of [native 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. In addition, YAML allows for private [types](#) specific to a single [application](#).

YAML was designed to support incremental interfaces that include both input ("**getNextEvent()**") and output ("**sendNextEvent()**") one-pass interfaces. Together, these enable YAML to support the processing of large [documents](#) (e.g. transaction logs) or continuous [streams](#) (e.g. feeds from a production machine).

1.3. Relation to JSON

Both JSON and YAML aim to be human readable data interchange formats. However, JSON and YAML have different priorities. JSON's foremost design goal is simplicity and universality. Thus, JSON is trivial to generate and parse, at the cost of reduced human readability. It also uses a lowest common denominator information model, ensuring any JSON data can be easily processed by every modern programming environment.

In contrast, YAML's foremost design goals are human readability and support for [serializing](#) arbitrary [native data structures](#). Thus, YAML allows for extremely readable files, but is more complex to generate and parse. In addition, YAML ventures beyond the lowest common denominator data types, requiring more complex processing when crossing between different programming environments.

YAML can therefore be viewed as a natural superset of JSON, offering improved human readability and a more complete information model. This is also the case in practice; every JSON file is also a valid YAML file. This makes it easy to migrate from JSON to YAML if/when the additional features are required.

JSON's [RFC4627](#) requires that [mappings keys](#) merely “SHOULD” be [unique](#), while YAML insists they “MUST” be. Technically, YAML therefore complies with the JSON spec, choosing to treat duplicates as an error. In practice, since JSON is silent on the semantics of such duplicates, the only portable JSON files are those with unique keys, which are therefore valid YAML files.

It may be useful to define an intermediate format between YAML and JSON. Such a format would be trivial to parse (but not very human readable), like JSON. At the same time, it would allow for [serializing](#) arbitrary [native data structures](#), like YAML. Such a format might also serve as YAML's "canonical format". Defining such a “YSON” format (YSON is a Serialized Object Notation) can be done either by enhancing the JSON specification or by restricting the YAML specification. Such a definition is beyond the scope of this specification.

1.4. Relation to XML

Newcomers to YAML often search for its correlation to the eXtensible Markup Language (XML). Although 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), which was designed to support structured documentation. XML therefore had many design constraints placed on it that YAML does not share. 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>.

1.5. Terminology

This specification uses key words based on [RFC2119](#) 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 to, 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.

The rest of this document is arranged as follows. Chapter [2](#) provides a short preview of the main YAML features. Chapter [3](#) describes the YAML information model, and the processes for converting from and to this model and the YAML text format. The bulk of the document, chapters [4](#) through [9](#), formally define this text format. Finally, chapter [10](#) recommends basic YAML schemas.

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 [key: value pair](#). [Comments](#) begin with an octothorpe (also called a "hash", "sharp", "pound", or "number sign" - `"#"`).

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      # Home runs
avg: 0.278  # Batting average
rbi: 147    # Runs Batted In
```

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 Example 2.6. Mapping of Mappings

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

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

2.2. Structures

YAML uses three dashes (“---”) to separate [directives](#) from [document content](#). This also serves to signal the start of a document if no [directives](#) are present. Three dots (“...”) indicate the end of a document without starting a new one, for use in communication channels.

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](#) (objects) 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. Compact 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](#) notation, using a [literal style](#) (indicated by “`|`”) where all [line breaks](#) are significant. Alternatively, they can be written with the [folded style](#) (denoted by “`>`”) 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 folded scalars, newlines become spaces

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

Example 2.15. Folded newlines are 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 two 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

Example 2.18. Multi-line Flow Scalars

unicode: "Sosa did fine.\u263A"
control: "\b1998\t1999\t2000\n"
hex esc: "\x0d\x0a is \r\n"

plain:
This unquoted scalar
spans many lines.

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

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

2.4. Tags

In YAML, [untagged nodes](#) are given a type depending on the [application](#). The examples in this specification generally use the [seq](#), [map](#) and [str](#) types from the [fail safe schema](#). A few examples also use the [int](#), [float](#), and [null](#) types from the [JSON schema](#). The [repository](#) includes additional types such as [binary](#), [omap](#), [set](#) and others.

Example 2.19. Integers

```
canonical: 12345
decimal: +12345
octal: 0o14
hexadecimal: 0xC
```

Example 2.20. Floating Point

```
canonical: 1.23015e+3
exponential: 12.3015e+02
fixed: 1230.15
negative infinity: -.inf
not a number: .NaN
```

Example 2.21. Miscellaneous

```
null:
booleans: [ true, false ]
string: '012345'
```

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 exclamation point (“[!](#)”) symbol. [Global tags](#) are URIs and may be specified in a [tag shorthand](#) notation 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
Pz7Y60juDg4J+fn50Tk6enp
```

Example 2.24. Global Tags

```
%TAG ! tag:clarkevans.com,200
--- !shape
# Use the ! handle for pres
# tag:clarkevans.com,2002:c
- !circle
center: &ORIGIN {x: 73, y:
radius: 7
```

<pre> 56enmleECcggoBADs= application specific tag: !something The semantics of the tag above may be different for different documents. </pre>	<pre> - !line start: *ORIGIN finish: { x: 89, y: 102 } - !label start: *ORIGIN color: 0xFFEEBB text: Pretty vector drawing </pre>
---	---

Example 2.25. Unordered Sets

```

# Sets are represented as a
# Mapping where each key is
# associated with a null value
--- !!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 [native 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.

3.1. Processes

Translating between [native data structures](#) and a character [stream](#) is done in several logically distinct stages, each with a well defined input and output data model, as shown in the following diagram:

Figure 3.1. Processing Overview



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](#). In particular, [mapping key order](#), [comments](#), and [tag handles](#) should not be referenced during [composition](#).

3.1.1. Dump

Dumping native data structures to a character [stream](#) is done using the following three stages:

Representing Native Data Structures

YAML *represents* any *native data structure* using three [node kinds](#): [sequence](#) - an ordered series of entries; [mapping](#) - an unordered association of [unique keys](#) to [values](#); and [scalar](#) - 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.

Serializing the Representation Graph

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.

Presenting the Serialization Tree

The final output process is *presenting* the YAML [serializations](#) as a character [stream](#) in a human-friendly manner. To maximize human readability, YAML offers 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 scalar 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 the [application](#), in general this process should be guided by the preferences of the user.

3.1.2. Load

Loading [native data structures](#) from a character [stream](#) is done using the following three stages:

Parsing the Presentation Stream

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 due to [ill-formed](#) input.

Composing the Representation Graph

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

Constructing Native Data Structures

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](#), [scalar 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.

The following diagram summarizes the three *information models*. Full arrows denote composition, hollow arrows denote inheritance, “1” and “*” denote “one” and “many” relationships. A single “+” denotes [serialization](#) details, a double “++” denotes [presentation](#) details.

Figure 3.2. Information Models



3.2.1. Representation Graph

YAML’s *representation* of [native data structure](#) 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](#)). 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](#); [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

A YAML *node* [represents](#) a single [native data structure](#). Such 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 the 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. 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 theoretical analysis and for creating practical YAML tools and APIs. This strategy is also used by the Javascript programming language.

3.2.1.2. Tags

YAML [represents](#) type information of [native data structures](#) with a simple

identifier, called a *tag*. *Global tags* are [URIs](#) and hence globally unique across all [applications](#). The “**tag:**” [URI scheme](#) 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 a 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. Node 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 scalar content](#). For example, the integer eleven can be written as “0013” (octal) or “0xB” (hexadecimal). If both notations 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 for producing the *canonical form* of any [formatted content](#). This form is a Unicode character string which also [presents](#) the same [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](#).

Different URI schemes may define different rules for testing the equality of URIs. Since a YAML [processor](#) cannot be reasonably expected to be aware of them all, it must resort to a simple character-by-character comparison of [tags](#) to ensure consistency. This also happens to be the comparison method defined by the “**tag:**” URI scheme. [Tags](#) in a YAML stream must therefore be [presented](#) in a canonical way so that such comparison would yield the correct results.

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 is 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 data structures](#) from the serial interface should not use [key order](#) or [anchor names](#) for the preservation of [application](#) 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 [application](#) 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*. 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 [styles](#), [scalar content 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 [serialization trees](#) to be contained in the same YAML character stream, as a series of [documents](#) separated by [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 styles](#) use [indentation](#) to denote structure; In contrast, [flow styles](#) rely on explicit [indicators](#).

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, [block sequences](#) and [mappings](#) begin on the next line. In some cases, YAML also allows nested [block collections](#) to start in-line for a more [compact notation](#). In addition, YAML provides a [compact notation](#) for [flow mappings](#) with a single [key: value pair](#), nested inside a [flow sequence](#). These allow for a natural “ordered mapping” notation.

Figure 3.6. Kind/Style Combinations



3.2.3.2. Scalar Formats

YAML allows [scalars](#) to be [presented](#) in several *formats*. For example, the integer “11” might also be written as “0xB”. [Tags](#) must specify a mechanism for converting the formatted 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 must 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 [formatted 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 [formatted 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 Streams and Identified Aliases

A [well-formed](#) character [stream](#) must match the BNF productions specified in the following chapters. 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 Tags

Typically, most [tags](#) are not explicitly specified in the character [stream](#). During [parsing](#), [nodes](#) lacking an explicit [tag](#) are given a *non-specific tag*: “!” for [non-plain scalars](#), and “?” for all other [nodes](#). [Composing](#) a [complete representation](#) requires each such non-specific tag to be *resolved* to a *specific tag*, be it a [global tag](#) or a [local tag](#).

Resolving the [tag](#) of a [node](#) must only depend on the following three parameters: (1) the non-specific tag of the [node](#), (2) the path leading from the [root](#) to the [node](#), and (3) the [content](#) (and hence the [kind](#)) of the [node](#). When a [node](#) has more than one occurrence (using [aliases](#)), tag resolution must depend only on the path to the first ([anchored](#)) occurrence of the [node](#).

Note that 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](#) to the resolved [node](#). Finally, resolution must not consider the [content](#) of a sibling [node](#) in a [collection](#), or the [content](#) of the [value node](#) associated with a [key node](#) being resolved.

These rules 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 referring to a relatively small number of previously parsed [nodes](#). Thus, in most cases, tag resolution in one-pass [processors](#) is both possible and practical.

YAML [processors](#) should resolve [nodes](#) having the “!” non-specific tag as “`tag:yaml.org,2002:seq`”, “`tag:yaml.org,2002:map`” or “`tag:yaml.org,2002:str`” depending on their [kind](#). This *tag resolution convention* allows the author of a YAML character [stream](#) to effectively “disable” the tag resolution process. By explicitly specifying a “!” non-specific [tag property](#), the [node](#) would then be resolved to a “vanilla” [sequence](#), [mapping](#), or string, according to its [kind](#).

[Application](#) specific tag resolution rules should be restricted to resolving the “?” non-specific tag, most commonly to resolving [plain scalars](#). These may be matched against a set of regular expressions to provide automatic resolution of integers, floats, timestamps, and similar types. An [application](#) may also match the [content](#) of [mapping nodes](#) against sets of expected [keys](#) to automatically resolve points, complex numbers, and similar types. Resolved [sequence node](#) types such as the “ordered mapping” are also possible.

That said, tag resolution is specific to the [application](#). YAML [processors](#) should therefore provide a mechanism allowing the [application](#) to override and expand these default tag resolution rules.

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](#) a [partial representation](#), based on each [node's kind](#) and allowing for non-specific tags.

3.3.3. Recognized and Valid Tags

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](#) cannot be used to [construct](#) a [native data structure](#).

3.3.4. Available Tags

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 Conventions

The following chapters formally define the syntax of YAML character [streams](#), using parameterized BNF productions. Each BNF production is both named and numbered for easy reference. Whenever possible, basic structures are specified before the more complex structures using them in a “bottom up” fashion.

The order of alternatives inside a production is significant. Subsequent alternatives are only considered when previous ones fails. See for example the [b-break](#) production. In addition, production matching is expected to be greedy. Optional (?), zero-or-more (*) and one-or-more (+) patterns are always expected to match as much of the input as possible.

The productions are accompanied by examples, which are given side-by-side next to equivalent YAML text in an explanatory format. This format uses only [flow collections](#), [double-quoted scalars](#), and explicit [tags](#) for each [node](#).

A reference implementation using the productions is available as the [YamlReference](#) Haskell package. This reference implementation is also available as an interactive web application at <http://dev.yaml.org/ypaste>.

4.1. Production Parameters

YAML's syntax is designed for maximal human readability. This requires [parsing](#) to depend on the surrounding text. For notational compactness, this dependency is expressed using parameterized BNF productions.

This context sensitivity is the cause of most of the complexity of the YAML syntax definition. It is further complicated by struggling with the human tendency to look ahead when interpreting text. These complications are of course the source of most of YAML's power to [present](#) data in a very human readable way.

Productions use any of the following parameters:

Indentation: n or m

Many productions use an explicit [indentation](#) level parameter. This is less elegant than Python's "indent" and "undent" conceptual tokens. However it is required to formally express YAML's indentation rules.

Context: c

This parameter allows productions to tweak their behavior according to their surrounding. YAML supports two groups of *contexts*, distinguishing between [block styles](#) and [flow styles](#).

In [block styles](#), [indentation](#) is used to delineate structure. To capture human perception of [indentation](#) the rules require special treatment of the [“-”](#) character, used in [block sequences](#). Hence in some cases productions need to behave differently inside [block sequences](#) (*block-in context*) and outside them (*block-out context*).

In [flow styles](#), explicit [indicators](#) are used to delineate structure. These styles can be viewed as the natural extension of JSON to cover [tagged](#), [single-quoted](#) and [plain scalars](#). Since the latter have no delineating [indicators](#), they are subject to some restrictions to avoid ambiguities. These restrictions depend on where they appear: as implicit keys directly inside a [block mapping](#) (*block-key*); as implicit keys inside a [flow mapping](#) (*flow-key*); as values inside a [flow collection](#) (*flow-in*); or as values outside one

(*flow-out*).

(Block) Chomping: t

Block scalars offer three possible mechanisms for [chomping](#) any trailing [line breaks](#): [strip](#), [clip](#) and [keep](#). Unlike the previous parameters, this only controls interpretation; the [line breaks](#) are valid in all cases.

4.2. Production Naming Conventions

To make it easier to follow production combinations, production names use a Hungarian-style naming convention. Each production is given a prefix based on the type of characters it begins and ends with.

e-

A production matching no characters.

c-

A production starting and ending with a special character.

b-

A production matching a single [line break](#).

nb-

A production starting and ending with a non-[break](#) character.

s-

A production starting and ending with a [white space](#) character.

ns-

A production starting and ending with a non-[space](#) character.

l-

A production matching complete line(s).

x-y-

A production starting with an x- character and ending with a y- character, where x- and y- are any of the above prefixes.

x+, x-y+

A production as above, with the additional property that the matched content [indentation](#) level is greater than the specified n parameter.

Chapter 5. Characters

5.1. Character Set

To ensure readability, YAML [streams](#) use only the *printable* subset of the Unicode character set. The allowed character range explicitly excludes the C0 control block `#x0-#x1F` (except for TAB `#x9`, LF `#xA`, and CR `#xD` which are allowed), DEL `#x7F`, the C1 control block `#x80-#x9F` (except for NEL `#x85` which is allowed), the surrogate block `#xD800-#xDFFF`, `#xFFFE`, and `#xFFFF`.

On input, a YAML [processor](#) must accept all Unicode characters except those explicitly excluded above.

On output, a YAML [processor](#) must only produce acceptable characters. Any excluded characters must be [presented](#) using [escape](#) sequences. In addition, any allowed characters known to be non-printable should also be [escaped](#). This isn't mandatory since a full implementation would require extensive character property tables.

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

To ensure [JSON compatibility](#), YAML [processors](#) must allow all non-control characters inside [quoted scalars](#). To ensure readability, non-printable characters should be [escaped](#) on output, even inside such [scalars](#). Note that JSON [quoted scalars](#) cannot span multiple lines or contain [tabs](#), but YAML [quoted scalars](#) can.

```
[2]          nb-json   ::=  #x9 | [#x20-#x10FFFF]
```

5.2. Character Encodings

All characters mentioned in this specification are Unicode code points. Each such code point is written as one or more bytes depending on the *character encoding* used. Note that in UTF-16, characters above `#xFFFF` are written as four bytes, using a surrogate pair.

The character encoding is a [presentation detail](#) and must not be used to convey [content](#) information.

On input, a YAML [processor](#) must support the UTF-8 and UTF-16 character encodings. For [JSON compatibility](#), the UTF-32 encodings must also be supported.

If a character [stream](#) begins with a *byte order mark*, the character encoding will be taken to be as indicated by the byte order mark. Otherwise, the [stream](#) must begin with an ASCII character. This allows the encoding to be deduced by the pattern of null (`#x00`) characters.

To make it easier to concatenate [streams](#), byte order marks may appear at the start of any [document](#). However all [documents](#) in the same [stream](#) must use the same character encoding.

To allow for [JSON compatibility](#), byte order marks are also allowed inside [quoted scalars](#). For readability, such [content](#) byte order marks should be [escaped](#) on output.

The encoding can therefore be deduced by matching the first few bytes of the [stream](#) with the following table rows (in order):

	Byte0	Byte1	Byte2	Byte3	Encoding
<i>Explicit BOM</i>	<code>#x00</code>	<code>#x00</code>	<code>#xFE</code>	<code>#xFF</code>	UTF-32BE
<i>ASCII first character</i>	<code>#x00</code>	<code>#x00</code>	<code>#x00</code>	<i>any</i>	UTF-32BE
<i>Explicit BOM</i>	<code>#xFF</code>	<code>#xFE</code>	<code>#x00</code>	<code>#x00</code>	UTF-32LE

<i>ASCII first character</i>	<i>any</i>	#x00	#x00	#x00	UTF-32LE
<i>Explicit BOM</i>	#xFE	#xFF			UTF-16BE
<i>ASCII first character</i>	#x00	<i>any</i>			UTF-16BE
<i>Explicit BOM</i>	#xFF	#xFE			UTF-16LE
<i>ASCII first character</i>	<i>any</i>	#x00			UTF-16LE
<i>Explicit BOM</i>	#xEF	#xBB	#xBF		UTF-8
<i>Default</i>					UTF-8

The recommended output encoding is UTF-8. If another encoding is used, it is recommended that an explicit byte order mark be used, even if the first [stream](#) character is ASCII.

For more information about the byte order mark and the Unicode character encoding schemes see the [Unicode FAQ](#).

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

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

Example 5.1. Byte Order Mark

↵# Comment only.

Legend:

[c-byte-order-mark](#)

This stream contains no
documents, only comments.

Example 5.2. Invalid Byte Order Mark

- Invalid use of BOM ERROR:
↵ A BOM must not appear

- Inside a document. inside a document.

5.3. Indicator Characters

Indicators are characters that have special semantics.

- | | | |
|-----|--------------------------|--|
| [4] | c-sequence-entry ::= “-” | A “-” (#x2D, hyphen) denotes a block sequence entry. |
| [5] | c-mapping-key ::= “?” | A “?” (#x3F, question mark) denotes a mapping key . |
| [6] | c-mapping-value ::= “:” | A “:” (#x3A, colon) denotes a mapping value . |

Example 5.3. Block Structure Indicators

```
sequence:                                %YAML 1.2
- one                                    ---
- two                                   !!map {
mapping:                                ? !!str "sequence"
  ? sky                                : !!seq [ !!str "one", !!str "two" ]
  : blue                               ? !!str "mapping"
  sea : green                          : !!map {
Legend:                                ? !!str "sky" : !!str "blue",
  c-sequence-entry                    ? !!str "sea" : !!str "green",
  c-mapping-key c-mapping-value      },
```

- | | | |
|------|--------------------------|---|
| [7] | c-collect-entry ::= “,” | A “,” (#x2C, comma) ends a flow collection entry. |
| [8] | c-sequence-start ::= “[” | A “[” (#x5B, left bracket) starts a flow sequence . |
| [9] | c-sequence-end ::= “]” | A “]” (#x5D, right bracket) ends a flow sequence . |
| [10] | c-mapping-start ::= “{” | A “{” (#x7B, left brace) starts a flow mapping . |
| [11] | c-mapping-end ::= “}” | A “}” (#x7D, right brace) ends a flow mapping . |

Example 5.4. Flow Collection Indicators

```
%YAML 1.2
---
sequence: [ one, two, ]
mapping: { sky: blue, sea: green }
Legend:
  c-sequence-start c-sequence-end
  c-mapping-start c-mapping-end
  c-collect-entry
!!map {
  ? !!str "sequence"
  : !!seq [ !!str "one", !!str "t
  ? !!str "mapping"
  : !!map {
    ? !!str "sky" : !!str "blue",
    ? !!str "sea" : !!str "green"
  },
}
```

- [12] c-comment ::= “#” An “<#>” (#x23, octothorpe, hash, sharp, pound, number sign) denotes a [comment](#).

Example 5.5. Comment Indicator

```
# Comment only.
Legend:
  c-comment
# This stream contains no
# documents, only comments.
```

- [13] c-anchor ::= “&” An “[&](#)” (#x26, ampersand) denotes a [node’s anchor property](#).
- [14] c-alias ::= “*” An “[*](#)” (#x2A, asterisk) denotes an [alias node](#).
- [15] c-tag ::= “!” The “[!](#)” (#x21, exclamation) is heavily overloaded for specifying [node tags](#). It is used to denote [tag handles](#) used in [tag directives](#) and [tag properties](#); to denote [local tags](#); and as the [non-specific tag](#) for non-[plain scalars](#).

Example 5.6. Node Property Indicators

```

                                %YAML 1.2
                                ---
anchored: !local &anchor value
alias: *anchor
Legend:
  c-tag c-anchor c-alias
                                !!map {
                                ? !!str "anchored"
                                : !local &A1 "value",
                                ? !!str "alias"
                                : *A1,
                                }

```

- [16] c-literal ::= “|” A “|” (#x7C, vertical bar) denotes a [literal block scalar](#).
- [17] c-folded ::= “>” A “>” (#x3E, greater than) denotes a [folded block scalar](#).

Example 5.7. Block Scalar Indicators

```

literal: |
  some
  text
folded: >
  some
  text
Legend:
  c-literal c-folded
                                %YAML 1.2
                                ---
                                !!map {
                                ? !!str "literal"
                                : !!str "some\ntext\n",
                                ? !!str "folded"
                                : !!str "some text\n",
                                }

```

- [18] c-single-quote ::= “'” An “'” (#x27, apostrophe, single quote) surrounds a [single-quoted flow scalar](#).
- [19] c-double-quote ::= “”” A “”” (#x22, double quote) surrounds a [double-quoted flow scalar](#).

Example 5.8. Quoted Scalar Indicators

```

single: 'text'
double: "text"
Legend:
  c-single-quote c-double-quote
                                %YAML 1.2
                                ---
                                !!map {
                                ? !!str "single"
                                : !!str "text",
                                ? !!str "double"

```



```

        : !!str "text",
    }

```

[20] c-directive ::= “%” A “%” (#x25, percent) denotes a [directive](#) line.

Example 5.9. Directive Indicator

```

%YAML 1.2
--- text      %YAML 1.2
Legend:      ---
             !!str "text"
             c-directive

```

[21] c-reserved ::= “@” | “`” The “@” (#x40, at) and “`” (#x60, grave accent) are *reserved* for future use.

Example 5.10. Invalid use of Reserved Indicators

```

commercial-at: @text
grave-accent: `text
ERROR:
Reserved indicators can't
start a plain scalar.

```

Any indicator character:

[22] c-indicator ::= [“_”](#) | [“?”](#) | [“.”](#) | [“_”](#) | [“\[”](#) | [“\]”](#) | [“{”](#) | [“}”](#)
| [“#”](#) | [“&”](#) | [“*”](#) | [“!”](#) | [“|”](#) | [“>”](#) | [“”](#) | [“”](#)
| [“%”](#) | [“@”](#) | [“`”](#)

The “[\[](#)”, “[\]](#)”, “[{](#)”, “[}](#)” and “[_](#)” indicators denote structure in [flow collections](#). They are therefore forbidden in some cases, to avoid ambiguity in several constructs. This is handled on a case-by-case basis by the relevant productions.

[23] c-flow-indicator ::= [“_”](#) | [“\[”](#) | [“\]”](#) | [“{”](#) | [“}”](#)

5.4. Line Break Characters

YAML recognizes the following ASCII *line break* characters.

- [24] b-line-feed ::= #xA /* LF */
- [25] b-carriage-return ::= #xD /* CR */
- [26] b-char ::= [b-line-feed](#) | [b-carriage-return](#)

All other characters, including the form feed (**#x0C**), are considered to be non-break characters. Note that these include the *non-ASCII line breaks*: next line (**#x85**), line separator (**#x2028**) and paragraph separator (**#x2029**).

[YAML version 1.1](#) did support the above non-ASCII line break characters; however, JSON does not. Hence, to ensure [JSON compatibility](#), YAML treats them as non-break characters as of version 1.2. In theory this would cause incompatibility with [version 1.1](#); in practice these characters were rarely (if ever) used. YAML 1.2 [processors parsing](#) a [version 1.1 document](#) should therefore treat these line breaks as non-break characters, with an appropriate warning.

- [27] nb-char ::= [c-printable](#) - [b-char](#) - [c-byte-order-mark](#)

Line breaks are interpreted differently by different systems, and have several widely used formats.

- [28] b-break ::= | ([b-carriage-return](#) [b-line-feed](#)) /* DOS, Windows */
 | [b-carriage-return](#) /* MacOS upto 9.x */
 | [b-line-feed](#) /* UNIX, MacOS X */

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 format is a [presentation detail](#) and must not be used to convey [content](#) information.

- [29] b-as-line-feed ::= [b-break](#)

Outside [scalar content](#), YAML allows any line break to be used to terminate

lines.

[30] `b-non-content ::= b-break`

On output, a YAML [processor](#) is free to emit line breaks using whatever convention is most appropriate.

In the examples, line breaks are sometimes displayed using the “↓” glyph for clarity.

Example 5.11. Line Break Characters

```
|
  Line break (no glyph)      %YAML 1.2
  Line break (glyphed)↓    ---
Legend:                    !!str "line break (no glyph)\n\
                             line break (glyphed)\n"
b-break
```

5.5. White Space Characters

YAML recognizes two *white space* characters: *space* and *tab*.

```
[31]          s-space   ::=  #x20 /* SP */
[32]          s-tab     ::=  #x9  /* TAB */
[33]          s-white   ::=  s-space | s-tab
```

The rest of the ([printable](#)) non-[break](#) characters are considered to be non-space characters.

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

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

Example 5.12. Tabs and Spaces

```
# Tabs and spaces                                %YAML 1.2
quoted:·"Quoted →"                               ---
block:→|                                         !!map {
··void main() {                                  ? !!str "quoted"
··→printf("Hello, world!\n");                   : "Quoted \t",
··}                                              ? !!str "block"
                                              : "void main() {\n\
                                              \tprintf(\"Hello, world!\\n\\");\n\
                                              }\n",
                                              }
Legend:
  s-space s-tab
```

5.6. Miscellaneous Characters

The YAML syntax productions make use of the following additional character classes:

- A decimal digit for numbers:

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

- A hexadecimal digit for [escape sequences](#):

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

- ASCII letter (alphabetic) characters:

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

- Word (alphanumeric) characters for identifiers:

[38] ns-word-char ::= [ns-dec-digit](#) | [ns-ascii-letter](#) | “-”

- URI characters for [tags](#), as specified in [RFC2396](#), with the addition of the “[” and “]” for presenting IPv6 addresses as proposed in [RFC2732](#).

By convention, any URI characters other than the allowed printable ASCII characters are first *encoded* in UTF-8, and then each byte is *escaped* using the “%” character. The YAML [processor](#) must not expand such escaped characters. [Tag](#) characters must be preserved and compared exactly as [presented](#) in the YAML [stream](#), without any processing.

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

- The “!” character is used to indicate the end of a [named tag handle](#); hence

its use in [tag shorthands](#) is restricted. In addition, such [shorthands](#) must not contain the “[\[](#)”, “[\]](#)”, “[{](#)”, “[}](#)” and “[.](#)” characters. These characters would cause ambiguity with [flow collection](#) structures.

[40] ns-tag-char ::= [ns-uri-char](#) - “[!](#)” - [c-flow-indicator](#)

5.7. Escaped Characters

All non-[printable](#) characters must be *escaped*. YAML escape sequences use the “\” notation common to most modern computer languages. Each escape sequence must be [parsed](#) into the appropriate Unicode character. The original escape sequence is a [presentation detail](#) and must not be used to convey [content](#) information.

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.

[41] c-escape ::= “\”

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

[42] ns-esc-null ::= “0” Escaped ASCII null (**#x0**) character.

[43] ns-esc-bell ::= “a” Escaped ASCII bell (**#x7**) character.

[44] ns-esc-backspace ::= “b” Escaped ASCII backspace (**#x8**) character.

[45] ns-esc-horizontal-tab ::= “t” |
tab #x9 Escaped ASCII horizontal tab (**#x9**) character. This is useful at the start or the end of a line to force a leading or trailing tab to become part of the [content](#).

[46] ns-esc-line-feed ::= “n” Escaped ASCII line feed (**#xA**) character.

[47] ns-esc-vertical-tab ::= “v” Escaped ASCII vertical tab (**#xB**) character.

[48] ns-esc-form-feed ::= “f” Escaped ASCII form feed (**#xC**) character.

[49] ns-esc-carriage-return ::= “r” Escaped ASCII carriage return (**#xD**) character.

::= Escaped ASCII escape (**#x1B**)

- [50] ns-esc-escape “e” character.
- [51] ns-esc-space ::= #x20 Escaped ASCII space (#x20) character. This is useful at the start or the end of a line to force a leading or trailing space to become part of the [content](#).
- [52] ns-esc-double-quote ::= [“”](#) Escaped ASCII double quote (#x22).
- [53] ns-esc-slash ::= “/” Escaped ASCII slash (#x2F), for [JSON compatibility](#).
- [54] ns-esc-backslash ::= [“\”](#) Escaped ASCII back slash (#x5C).
- [55] ns-esc-next-line ::= “N” Escaped Unicode next line (#x85) character.
- [56] ns-esc-non-breaking-space ::= “ ” Escaped Unicode non-breaking space (#xA0) character.
- [57] ns-esc-line-separator ::= “L” Escaped Unicode line separator (#x2028) character.
- [58] ns-esc-paragraph-separator ::= “p” Escaped Unicode paragraph separator (#x2029) character.
- [59] ns-esc-8-bit ::= ([“x”](#) [ns-hex-digit](#) × 2) Escaped 8-bit Unicode character.
- [60] ns-esc-16-bit ::= ([“u”](#) [ns-hex-digit](#) × 4) Escaped 16-bit Unicode character.
- [61] ns-esc-32-bit ::= ([“U”](#) [ns-hex-digit](#) × 8) Escaped 32-bit Unicode character.

Any escaped character:

[“\”](#)
([ns-esc-null](#) | [ns-esc-bell](#) | [ns-esc-backspace](#)

	ns-esc-horizontal-tab ns-esc-line-feed
[62] c-ns-esc-char : :=	ns-esc-vertical-tab ns-esc-form-feed
	ns-esc-carriage-return ns-esc-escape ns-esc-space
	ns-esc-double-quote ns-esc-slash 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 5.13. Escaped Characters

"Fun with \\ \" \a \b \e \f \\ \n \r \t \v \0 \\ \ _ \N \L \P \\ \x41 \u0041 \U00000041"	%YAML 1.2 --- "Fun with \x5C \x22 \x07 \x08 \x1B \x0C \x0A \x0D \x09 \x0B \x00 \x20 \xA0 \x85 \u2028 \u2029 A A A"
---	--

Legend:
[c-ns-esc-char](#)

Example 5.14. Invalid Escaped Characters

Bad escapes: "\c \xq-	ERROR: - c is an invalid escaped character - q and - are invalid hex digits.
-----------------------------	--

Chapter 6. Basic Structures

6.1. Indentation Spaces

In YAML [block styles](#), structure is determined by *indentation*. In general, indentation is defined as a zero or more [space](#) characters at the start of a line.

To maintain portability, [tab](#) characters must not be used in indentation, 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](#).

The amount of indentation is a [presentation detail](#) and must not be used to convey [content](#) information.

[63] $\text{s-indent}(n) ::= \text{s-space} \times n$

A [block style](#) construct is terminated when encountering a line which is less indented than the construct. The productions use the notation “**s-indent(<n)**” and “**s-indent(≤n)**” to express this.

[64] $\text{s-indent}(<n) ::= \text{s-space} \times m \text{ /* Where } m < n \text{ */}$

[65] $\text{s-indent}(\leq n) ::= \text{s-space} \times m \text{ /* Where } m \leq n \text{ */}$

Each [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.

Example 6.1. Indentation Spaces

```
..# Leading comment line spaces are %YAML 1.2
...# neither content nor indentation. - - -
....
Not indented: !!map {
  .By one space: | ? !!str "Not indented"
  ....By four : !!map {
  .....spaces ? !!str "By one space"
  .Flow style: [ : !!str "By four\n spac
  ...By two, # in flow style ? !!str "Flow style"
  ..Also by two, # are neither : !!seq [
  ..→Still by two # content nor !!str "By two",
```

```

.....]          # indentation.          !!str "Also by two",
                                           !!str "Still by two"
Legend:
  s-indent\(n\) Content          }
  Neither content nor indentation }

```

The “-”, “?” and “:” characters used to denote [block collection](#) entries are perceived by people to be part of the indentation. This is handled on a case-by-case basis by the relevant productions.

Example 6.2. Indentation Indicators

```

?·a                                     %YAML 1.2
:·-→b                                 ---
:·-·-·-→c                             !!map {
:·-·-·-·d                             ? !!str "a"
                                           : !!seq [
                                           !!str "b",
                                           !!seq [ !!str "c", !!str "d"
Legend:
  Total Indentation
  s-indent\(n\) Indicator as indentation ],

```

6.2. Separation Spaces

Outside [indentation](#) and [scalar content](#), YAML uses [white space](#) characters for *separation* between tokens within a line. Note that such [white space](#) may safely include [tab](#) characters.

Separation spaces are a [presentation detail](#) and must not be used to convey [content](#) information.

[66] s-separate-in-line ::= [s-white](#)+ | /* Start of line */

Example 6.3. Separation Spaces

<pre>- .foo:→.bar - - .baz -→baz</pre> <p>Legend: s-separate-in-line</p>	<pre>%YAML 1.2 --- !!seq [!!map { ? !!str "foo" : !!str "bar", }, !!seq [!!str "baz", !!str "baz"],]</pre>
--	--

6.3. Line Prefixes

Inside [scalar content](#), each line begins with a non-[content](#) *line prefix*. This prefix always includes the [indentation](#). For [flow scalar styles](#) it additionally includes all leading [white space](#), which may contain [tab](#) characters.

Line prefixes are a [presentation detail](#) and must not be used to convey [content](#) information.

[67] $c = \text{block-out} \Rightarrow \text{s-block-line-prefix}(n)$
 $c = \text{block-in} \Rightarrow \text{s-block-line-prefix}(n)$
 $c = \text{flow-out} \Rightarrow \text{s-flow-line-prefix}(n)$
 $c = \text{flow-in} \Rightarrow \text{s-flow-line-prefix}(n)$
[68] $\text{s-block-line-prefix}(n) ::= \text{s-indent}(n)$
[69] $\text{s-flow-line-prefix}(n) ::= \text{s-indent}(n) \text{ s-separate-in-line?}$

Example 6.4. Line Prefixes

plain: text	%YAML 1.2
..lines	---
quoted: "text	!!map {
..→lines"	? !!str "
block:	: !!str "
..text	? !!str "
...→lines	: !!str "
	? !!str "
	: !!str "

Legend:
 [s-flow-line-prefix\(n\)](#) [s-block-line-prefix\(n\)](#) [s-indent\(n\)](#)

6.4. Empty Lines

An *empty line* consists of the non-[content prefix](#) followed by a [line break](#).

[70] $\text{l-empty}(n,c) ::= (\text{s-line-prefix}(n,c) \mid \text{s-indent}(<n))$
[b-as-line-feed](#)

The semantics of empty lines depend on the [scalar style](#) they appear in. This is handled on a case-by-case basis by the relevant productions.

Example 6.5. Empty Lines

Folding:	%YAML 1.2
"Empty line	---
...→	!!map {
as a line feed"	? !!str "Folding"
Chomping:	: !!str "Empty line\nas a line feed",
Clipped empty lines	? !!str "Chomping"
.	: !!str "Clipped empty lines\n",
Legend:	}
l-empty(n,c)	

6.5. Line Folding

Line folding allows long lines to be broken for readability, while retaining the semantics of the original long line. If a [line break](#) is followed by an [empty line](#), it is *trimmed*; the first [line break](#) is discarded and the rest are retained as [content](#).

[71] $\text{b-l-trimmed}(n,c) ::= \text{b-non-content l-empty}(n,c)^+$

Otherwise (the following line is not [empty](#)), the [line break](#) is converted to a single [space](#) (`#x20`).

[72] $\text{b-as-space} ::= \text{b-break}$

A folded non-[empty line](#) may end with either of the above [line breaks](#).

[73] $\text{b-l-folded}(n,c) ::= \text{b-l-trimmed}(n,c) \mid \text{b-as-space}$

Example 6.6. Line Folding

<pre>>- trimmed↓ ..↓ .↓ ↓ as↓ space</pre>	<pre>%YAML 1.2 --- !!str "trimmed\n\n\nas space" Legend: b-l-trimmed(n,c) b-as-space</pre>
--	---

The above rules are common to both the [folded block style](#) and the [scalar flow styles](#). Folding does distinguish between these cases in the following way:

Block Folding

In the [folded block style](#), the final [line break](#) and trailing [empty lines](#) are subject to [chomping](#), and are never folded. In addition, folding does not apply to [line breaks](#) surrounding text lines that contain leading [white space](#). Note that such a [more-indented](#) line may consist only of such leading [white](#)

space.

The combined effect of the *block line folding* rules is that each “paragraph” is interpreted as a line, [empty lines](#) are interpreted as a line feed, and the formatting of [more-indented](#) lines is preserved.

Example 6.7. Block Folding

```
>
..foo.↓
.↓
.→.bar↓
↓
..baz↓

%YAML 1.2
--- !!str
"foo \n\n\t bar\n\nbaz\n"

Legend:
  b-1-folded(n,c)
    Non-content spaces Content spaces
```

Flow Folding

Folding in [flow styles](#) provides more relaxed semantics. [Flow styles](#) typically depend on explicit [indicators](#) rather than [indentation](#) to convey structure. Hence spaces preceding or following the text in a line are a [presentation detail](#) and must not be used to convey [content](#) information. Once all such spaces have been discarded, all [line breaks](#) are folded, without exception.

The combined effect of the *flow line folding* rules is that each “paragraph” is interpreted as a line, [empty lines](#) are interpreted as line feeds, and text can be freely [more-indented](#) without affecting the [content](#) information.

[74] $\text{s-flow-folded}(n) ::= \frac{\text{s-separate-in-line? b-l-folded}(n, \text{flow-in})}{\text{s-flow-line-prefix}(n)}$

Example 6.8. Flow Folding

```
"↓                                %YAML 1.2
· ·foo·↓                        --- !!str
·↓                              " foo\nbar\nbaz "
· ·→·bar↓
↓                                Legend:
```

· · baz ↓
"

[s-flow-folded\(n\)](#)
Non-content spaces

6.6. Comments

An explicit *comment* is marked by a “#” *indicator*. Comments are a [presentation detail](#) and must not be used to convey [content](#) information.

Comments must be [separated](#) from other tokens by [white space](#) characters. To ensure [JSON compatibility](#), YAML [processors](#) must allow for the omission of the final comment [line break](#) of the input [stream](#). However, as this confuses many tools, YAML [processors](#) should terminate the [stream](#) with an explicit [line break](#) on output.

```

[75] c-nb-comment-text ::= "#" nb-char*
[76] b-comment ::= b-non-content | /* End of file */
[77] s-b-comment ::= ( s-separate-in-line c-nb-comment-text? )?
                        b-comment

```

Example 6.9. Separated Comment

```
key:....# Comment↓                                %YAML 1.2
  valueeof                                          ---
                                                    !!map {
Legend:                                             ? !!str "key"
  c-nb-comment-text b-comment                  : !!str "value",
  s-b-comment                                     }
```

Outside [scalar content](#), comments may appear on a line of their own, independent of the [indentation](#) level. Note that outside [scalar content](#), a line containing only [white space](#) characters is taken to be a comment line.

[78] l-comment ::= [s-separate-in-line](#) [c-nb-comment-text](#)? [b-comment](#)

Example 6.10. Comment Lines

```

# This stream contains no
..# Comment↓ # documents, only comments.

```

...↓
↓

Legend:
[s-b-comment](#) [l-comment](#)

In most cases, when a line may end with a comment, YAML allows it to be followed by additional comment lines. The only exception is a comment ending a [block scalar header](#).

[79] s-l-comments ::= ([s-b-comment](#) | /* Start of line */)
[l-comment](#)*

Example 6.11. Multi-Line Comments

```

                                %YAML 1.2
key:.....# Comment↓ ---
.....# lines↓    !!map {
  value↓          ? !!str "key"
↓                : !!str "value",
                  }

```

Legend:
[s-b-comment](#) [l-comment](#) [s-l-comments](#)

6.7. Separation Lines

[Implicit keys](#) are restricted to a single line. In all other cases, YAML allows tokens to be separated by multi-line (possibly empty) [comments](#).

Note that structures following multi-line comment separation must be properly [indented](#), even though there is no such restriction on the separation [comment](#) lines themselves.

```
[80]      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 = block-key ⇒ s-separate-in-line
                                c = flow-key  ⇒ s-separate-in-line
[81]      s-separate-lines(n) ::= ( s-l-comments s-flow-line-prefix\(n\) )
                                | s-separate-in-line
```

Example 6.12. Separation Spaces

```
%YAML 1.2
---
{.first:.Sammy,.last:.Sosa.}:↓ !!map {
# Statistics:                ? !!map {
  .hr:..# Home runs          ? !!str "first"
  .....65                   : !!str "Sammy",
  .avg:..# Average           ? !!str "last"
  ...0.278                   : !!str "Sosa",
                             }
                             : !!map {
                             ? !!str "hr"
                             : !!int "65",
                             ? !!str "avg"
                             : !!float "0.278",
                             },
}
```

Legend:
[s-separate-in-line](#)
[s-separate-lines\(n\)](#)
[s-indent\(n\)](#)

6.8. Directives

Directives are instructions to the YAML [processor](#). 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.

Directives are a [presentation detail](#) and must not be used to convey [content](#) information.

```
[82]          l-directive ::= “%”
                               ( ns-yaml-directive
                               | ns-tag-directive
                               | ns-reserved-directive )
                               s-l-comments
```

Each directive is specified on a separate non-[indented](#) line starting with the [“%”](#) *indicator*, followed by the directive name and a list of parameters. The semantics of these parameters depends on the specific directive. A YAML [processor](#) should ignore unknown directives with an appropriate warning.

```
[83] ns-reserved-directive ::= ns-directive-name
                               ( s-separate-in-line ns-directive-parameter )*
[84] ns-directive-name ::= ns-char+
[85] ns-directive-parameter ::= ns-char+
```

Example 6.13. Reserved Directives

```
%F00 bar baz # Should be ignored%YAML 1.2
              # with a warning. --- !!str
--- "foo"          "foo"
```

Legend:

[ns-reserved-directive](#) [ns-directive-name](#) [ns-directive-parameter](#)

6.8.1. “YAML” Directives

The “**YAML**” *directive* specifies the version of YAML the [document](#) conforms to. This specification defines version “**1.2**”, including recommendations for *YAML 1.1 processing*.

A version 1.2 YAML [processor](#) must accept [documents](#) with an explicit “%YAML 1.2” directive, as well as [documents](#) lacking a “YAML” directive. Such [documents](#) are assumed to conform to the 1.2 version specification. [Documents](#) with a “YAML” directive specifying a higher minor version (e.g. “%YAML 1.3”) 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.

A version 1.2 YAML [processor](#) must also accept [documents](#) with an explicit “%YAML 1.1” directive. Note that version 1.2 is mostly a superset of version 1.1, defined for the purpose of ensuring *JSON compatibility*. Hence a version 1.2 [processor](#) should process version 1.1 [documents](#) as if they were version 1.2, giving a warning on points of incompatibility (handling of [non-ASCII line breaks](#), as described [above](#)).

[86] ns-yaml-directive ::= “Y” “A” “M” “L”
[s-separate-in-line](#) [ns-yaml-version](#)

[87] ns-yaml-version ::= [ns-dec-digit](#)+ “.” [ns-dec-digit](#)+ “.”

Example 6.14. “YAML” directive

```
%YAML 1.3 # Attempt parsing
          # with a warning
---
"foo"

%YAML 1.2
---
!!str "foo"

Legend:
  ns-yaml-directive ns-yaml-versi
```

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

Example 6.15. Invalid Repeated YAML directive

```
%YAML 1.2
%YAML 1.1
foo
```

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

6.8.2. “TAG” Directives

The “**TAG**” *directive* establishes a [tag shorthand](#) notation for specifying [node tags](#). Each “**TAG**” directive associates a [handle](#) with a [prefix](#). This allows for compact and readable [tag](#) notation.

```
[88]          ns-tag-directive ::= “T” “A” “G”
s-separate-in-line c-tag-handle
s-separate-in-line ns-tag-prefix
```

Example 6.16. “TAG” directive

```
%TAG !yaml! tag:yaml.org,2002: %YAML 1.2
---
!yaml!str "foo"                ---
!!str "foo"
```

Legend:

[ns-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 occurrences give the same [prefix](#).

Example 6.17. 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.
```


6.8.2.1. Tag Handles

The *tag handle* exactly matches the prefix of the affected [tag shorthand](#). There are three tag handle variants:

```
[89]          c-tag-handle ::= | c-named-tag-handle  
                                     | c-secondary-tag-handle  
                                     | c-primary-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 the default 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 the simple addition of a single “TAG” directive.

```
[90]          c-primary-tag-handle ::= “!”
```

Example 6.18. Primary Tag Handle

```
%YAML 1.2  
# Private  
!foo "bar"  
...  
# Global  
%TAG ! tag:example.com,2000:app/ <tag:example.com,2000:app/foo> "bar"  
...  
!foo "bar"  
  
Legend:  
c-primary-tag-handle
```

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:**”. This prefix is used by the [YAML tag repository](#).

It is possible to override this default behavior by providing an explicit **“TAG”** directive associating a different prefix for this handle.

[91] c-secondary-tag-handle ::= “! “!

Example 6.19. Secondary Tag Handle

```
%TAG !! tag:example.com,2000:app/
---
!!int 1 - 3 # Interval, not integer %YAML 1.2
---
!<tag:example.com,2000:app/int>
```

Legend:

[c-secondary-tag-handle](#)

Named Handles

A *named tag handle* surrounds a non-empty name with “!” characters. A handle name must not be used in a [tag shorthand](#) unless an explicit “**TAG**” directive has associated some prefix with it.

The name of the handle is a [presentation detail](#) and must not be used to convey [content](#) information. In particular, the YAML [processor](#) need not preserve the handle name once [parsing](#) is completed.

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

Example 6.20. Tag Handles

[illegible]

[c-named-tag-handle](#)

6.8.2.2. Tag Prefixes

There are two *tag prefix* variants:

[93] ns-tag-prefix ::= [c-ns-local-tag-prefix](#) | [ns-global-tag-prefix](#)

Local Tag Prefix

If the prefix begins with a “[!](#)” character, [shorthands](#) using the [handle](#) are expanded to a [local tag](#). Note that such a [tag](#) is intentionally not a valid URI, and its semantics are specific to the [application](#). In particular, two [documents](#) in the same [stream](#) may assign different semantics to the same [local tag](#).

[94] c-ns-local-tag-prefix ::= “[!](#)” [ns-uri-char](#)*

Example 6.21. Local Tag Prefix

```
%TAG !m! !my-
--- # Bulb here
!m!light fluorescent
...
%TAG !m! !my-
--- # Color here
!m!light green
Legend:
c-ns-local-tag-prefix
```

%YAML 1.2	!<!my-light> "fluorescent"
%YAML 1.2	!<!my-light> "green"

Global Tag Prefix

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, every [documents](#) in every [stream](#) must assign the same semantics to the

same [global tag](#).

[95] ns-global-tag-prefix ::= [ns-tag-char](#) [ns-uri-char](#)*

Example 6.22. Global Tag Prefix

```
%TAG !e! tag:example.com,2000:app/
```

```
---
```

```
- !e!foo "bar"
```

Legend:

[ns-global-tag-prefix](#)

```
%YAML 1.2
```

```
---
```

```
!<tag:example.com,2000:app/foo> "
```

6.9. Node Properties

Each [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](#). Either or both may be omitted.

[96]
$$\text{c-ns-properties}(n,c) ::= \begin{array}{l} (\text{c-ns-tag-property} \\ (\text{s-separate}(n,c) \text{ c-ns-anchor-property})?) \\ | (\text{c-ns-anchor-property} \\ (\text{s-separate}(n,c) \text{ c-ns-tag-property})?) \end{array}$$

Example 6.23. Node Properties

```
!!str &a1 "foo":      %YAML 1.2
  !!str bar         ---
&a2 baz : *a1        !!map {
Legend:              ? &B1 !!str "foo"
  c-ns-properties(n,c) : !!str "bar",
  c-ns-anchor-property ? !!str "baz"
  c-ns-tag-property    : *B1,
                       }
```

6.9.1. Node Tags

The *tag property* identifies the type of the [native data structure presented](#) by the [node](#). A tag is denoted by the “!” *indicator*.

[97]
$$\text{c-ns-tag-property} ::= \begin{array}{l} \text{c-verbatim-tag} \\ | \text{c-ns-shorthand-tag} \\ | \text{c-non-specific-tag} \end{array}$$

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](#)).

[98] c-verbatim-tag ::= [“!”](#) [“<”](#) [ns-uri-char](#)+ [“>”](#)

Example 6.24. Verbatim Tags

```
!<tag:yaml.org,2002:str> foo :    %YAML 1.2
  !<!bar> baz                    ---
                                !!map {
Legend:                          ? !<tag:yaml.org,2002:str> "foo"
  c-verbatim-tag                : !<!bar> "baz",
                                }
                                }
```

Example 6.25. 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](#)).

The choice of [tag handle](#) is a [presentation detail](#) and must not be used to convey [content](#) information. In particular, the [tag handle](#) may be discarded once [parsing](#) is completed.

The suffix must not contain any [“!”](#) character. This would cause the tag shorthand to be interpreted as having a [named tag handle](#). In addition, the suffix must not contain the [“\[”](#), [“\]”](#), [“{”](#), [“}”](#) and [“.”](#) characters. These

characters would cause ambiguity with [flow collection](#) structures. If the suffix needs to specify any of the above restricted characters, they must be [escaped](#) using the “%” character. This behavior is consistent with the URI character escaping rules (specifically, section 2.3 of [RFC2396](#)).

[99] c-ns-shorthand-tag ::= [c-tag-handle](#) [ns-tag-char](#)⁺

Example 6.26. Tag Shorthands

```
%TAG !e! tag:example.com,2000:app/ %YAML 1.2
---
- !local foo
- !!str bar
- !e!tag%21 baz
Legend:
c-ns-shorthand-tag
```

```
---
!!seq [
  !<!local> "foo",
  !<tag:yaml.org,2002:str> "bar",
  !<tag:example.com,2000:app/tag!
]
```

Example 6.27. Invalid Tag Shorthands

```
%TAG !e! tag:example,2000:app/
---
- !e! foo
- !h!bar baz
```

ERROR:

- 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](#) that needs to be [resolved](#) to a [specific](#) one. This [non-specific tag](#) is “!” for non-[plain scalars](#) and “?” for all other [nodes](#). This is the only case where the [node style](#) has any effect on the [content](#) information.

It is possible for the tag property to be explicitly set to the “!” [non-specific tag](#). By [convention](#), this “disables” [tag resolution](#), forcing the [node](#) to be interpreted as “tag:yaml.org,2002:seq”, “tag:yaml.org,2002:map”, or “tag:yaml.org,2002:str”, according to its [kind](#).

There is no way to explicitly specify the “?” [non-specific](#) tag. This is intentional.

[100] c-non-specific-tag ::= “!”

Example 6.28. Non-Specific Tags

```
# Assuming conventional resolution: %YAML 1.2
- "12"          ---
- 12            !!seq [
- ! 12          !<tag:yaml.org,2002:str> "12",
                  !<tag:yaml.org,2002:int> "12",
Legend:         !<tag:yaml.org,2002:str> "12",
  c-non-specific-tag ]
```

6.9.2. Node Anchors

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

[101] 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](#).

Anchor names must not contain the “[”, “]”, “{”, “}” and “.” characters. These characters would cause ambiguity with [flow collection](#) structures.

[102] ns-anchor-char ::= [ns-char](#) - [c-flow-indicator](#)

[103] ns-anchor-name ::= [ns-anchor-char](#)⁺

Example 6.29. Node Anchors


```

%YAML 1.2
---
First occurrence: &anchor Value
Second occurrence: *anchor
Legend:
  c-ns-anchor-property ns-anchor-name
  ? !!str "First occurrence"
  : &A !!str "Value",
  ? !!str "Second occurrence"
  : *A,
  }

```

Chapter 7. Flow Styles

YAML's *flow styles* can be thought of as the natural extension of JSON to cover [folding](#) long content lines for readability, [tagging](#) nodes to control [construction](#) of [native data structures](#), and using [anchors](#) and [aliases](#) to reuse [constructed](#) object instances.

7.1. Alias Nodes

Subsequent occurrences of a previously [serialized](#) node are [presented](#) as *alias nodes*. 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 is denoted by the “*” *indicator*. The alias refers to the most recent preceding [node](#) having the same [anchor](#). It is an error for an alias node to 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](#).

[104] c-ns-alias-node ::= [“*”](#) [ns-anchor-name](#)

Example 7.1. Alias Nodes

	%YAML 1.2

First occurrence: &anchor Foo	!!map {
Second occurrence: *anchor	? !!str "First occurrence"
Override anchor: &anchor Bar	: &A !!str "Foo",
Reuse anchor: *anchor	? !!str "Override anchor"
	: &B !!str "Bar",
Legend:	? !!str "Second occurrence"
c-ns-alias-node ns-anchor-name	: *A,
	? !!str "Reuse anchor"
	: *B,
	}

7.2. Empty Nodes

YAML allows the [node content](#) to be omitted in many cases. [Nodes](#) with empty [content](#) are interpreted as if they were [plain scalars](#) with an empty value. Such [nodes](#) are commonly resolved to a [“null”](#) value.

[105] e-scalar ::= /* Empty */

In the examples, empty [scalars](#) are sometimes displayed as the glyph “°” for clarity. Note that this glyph corresponds to a position in the characters [stream](#) rather than to an actual character.

Example 7.2. Empty Content

```
{
  foo : !!str°,
  !!str° : bar,
}
Legend:
  e-scalar
```

```
%YAML 1.2
---
!!map {
  ? !!str "foo" : !!str "",
  ? !!str ""    : !!str "bar",
}
```

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 indication for their existence.

[106] e-node ::= [e-scalar](#)

Example 7.3. Completely Empty Flow Nodes

```
{
  ? foo : °,
  °: bar,
}
Legend:
  e-node
```

```
%YAML 1.2
---
!!map {
  ? !!str "foo" : !!null "",
  ? !!null ""   : !!str "bar",
}
```

7.3. Flow Scalar Styles

YAML provides three *flow scalar styles*: [double-quoted](#), [single-quoted](#) and [plain](#) (unquoted). Each provides a different trade-off between readability and expressive power.

The [scalar style](#) is a [presentation detail](#) and must not be used to convey [content](#) information, with the exception that [plain scalars](#) are distinguished for the purpose of [tag resolution](#).

7.3.1. Double-Quoted Style

The *double-quoted style* is specified by surrounding “” indicators. This is the only [style](#) capable of expressing arbitrary strings, by using [“\” escape sequences](#). This comes at the cost of having to escape the [“\”](#) and “” characters.

[107] `nb-double-char ::= c-ns-esc-char | (nb-json - “\” - “”)`

[108] ns-double-char ::= nb-double-char - s-white

Double-quoted scalars are restricted to a single line when contained inside an [implicit key](#).

[109] $\text{c-double-quoted}(n,c) ::= \text{“” nb-double-text}(n,c) \text{””}$
 $\text{c} = \text{flow-out} \Rightarrow \text{nb-double-multi-line}(n)$

[110] $\text{nb-double-text}(n,c) ::=$ $c = \text{flow-in} \Rightarrow \text{nb-double-multi-line}(n)$
 $c = \text{block-key} \Rightarrow \text{nb-double-one-line}$
 $c = \text{flow-key} \Rightarrow \text{nb-double-one-line}$

[111] nb-double-one-line ::= nb-double-char*

Example 7.4. Double Quoted Implicit Keys

```

%YAML 1.2
---
"implicit block key" : [
  "implicit flow key" : value,
]
!!map {
  ? !!str "implicit block key"
  : !!seq [
    !!map {

```

Legend:

```

? !!str "implicit flow key"
nb-double-one-line      : !!str "value",
c-double-quoted(n,c)   }
                        ]
                        }

```

In a multi-line double-quoted scalar, [line breaks](#) are subject to [flow line folding](#), which discards any trailing [white space](#) characters. It is also possible to *escape* the [line break](#) character. In this case, the [line break](#) is excluded from the [content](#), and the trailing [white space](#) characters are preserved. Combined with the ability to [escape white space](#) characters, this allows double-quoted lines to be broken at arbitrary positions.

[112] s-double-escaped(n) ::= [s-white](#)* [“\”](#) [b-non-content](#)
[l-empty\(n,flow-in\)](#)* [s-flow-line-prefix\(n\)](#)

[113] s-double-break(n) ::= [s-double-escaped\(n\)](#) | [s-flow-folded\(n\)](#)

Example 7.5. Double Quoted Line Breaks

```

"folded·↓                                %YAML 1.2
to a space,→↓                            ---
·↓                                        !!str "folded to a space,\n\
to a line feed, or·→\↓                    to a line feed, \
·\·→non-content"                        or \t \tnon-content"

```

Legend:

[s-flow-folded\(n\)](#) [s-double-escaped\(n\)](#)

All leading and trailing [white space](#) characters are excluded from the [content](#). Each continuation line must therefore contain at least one non-[space](#) character. Empty lines, if any, are consumed as part of the [line folding](#).

[114] nb-ns-double-in-line ::= ([s-white](#)* [ns-double-char](#))*

[s-double-break\(n\)](#)
[115] s-double-next-line(n) ::= ([ns-double-char](#) [nb-ns-double-in-line](#)
([s-double-next-line\(n\)](#) | [s-white](#)*))?

[116] nb-double-multi-line(n) ::= nb-ns-double-in-line
 (s-double-next-line(n) | s-white^{*})

Example 7.6. Double Quoted Lines

```
"·1st non-empty↓
↓
·2nd non-empty·
→3rd non-empty·"

%YAML 1.2
---
!!str " 1st non-empty\n\
      2nd non-empty \
      3rd non-empty "
```

Legend:

nb-ns-double-in-line s-double-next-line(n)

7.3.2. Single-Quoted Style

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. In addition, it is only possible to break a long single-quoted line where a [space](#) character is surrounded by non-[spaces](#).

```
[117]      c-quoted-quote ::= """ """
[118]      nb-single-char ::= c-quoted-quote | ( nb-json - """ )
[119]      ns-single-char ::= nb-single-char - s-white
```

Example 7.7. Single Quoted Characters

```
'here''s to "quotes"'      %YAML 1.2
Legend:                    ---
  c-quoted-quote          !!str "here's to \"quotes\""

```

Single-quoted scalars are restricted to a single line when contained inside a

implicit key.

[120] c-single-quoted(n,c) ::= “” nb-single-text(n,c) “”
c = flow-out ⇒ nb-single-multi-line(n)
[121] nb-single-text(n,c) ::= c = flow-in ⇒ nb-single-multi-line(n)
c = block-key ⇒ nb-single-one-line
c = flow-key ⇒ nb-single-one-line
[122] nb-single-one-line ::= nb-single-char*

Example 7.8. Single Quoted Implicit Keys

```
%YAML 1.2
---
'implicit block key' : [
  'implicit flow key' : value,
]

Legend:
  nb-single-one-line
  c-single-quoted\(n,c\)
```

All leading and trailing [white space](#) characters are excluded from the [content](#). Each continuation line must therefore contain at least one non-[space](#) character. Empty lines, if any, are consumed as part of the [line folding](#).

[123] $\text{nb-ns-single-in-line} ::= (\text{s-white}^* \text{ ns-single-char})^* \text{s-flow-folded}(n)$

[124] $\text{s-single-next-line}(n) ::= (\text{ns-single-char } \text{nb-ns-single-in-line} (\text{s-single-next-line}(n) \mid \text{s-white}^*))?$

[125] $\text{nb-single-multi-line}(n) ::= \text{nb-ns-single-in-line} (\text{s-single-next-line}(n) \mid \text{s-white}^*)$

Example 7.9. Single Quoted Lines

```
' .1st non-empty↓                                %YAML 1.2
```



```

↓
·2nd non-empty·          ---
→3rd non-empty·'        !!str " 1st non-empty\n\
                           2nd non-empty \
                           3rd non-empty "

```

Legend:

[nb-ns-single-in-line\(n\)](#) [s-single-next-line\(n\)](#)

7.3.3. Plain Style

The *plain* (unquoted) style has no identifying [indicators](#) and provides no form of escaping. It is therefore the most readable, most limited and most [context sensitive style](#). In addition to a restricted character set, a plain scalar must not be empty, or contain leading or trailing [white space](#) characters. It is only possible to break a long plain line where a [space](#) character is surrounded by non-[spaces](#).

Plain scalars must not begin with most [indicators](#), as this would cause ambiguity with other YAML constructs. However, the “:”, “?” and “-” [indicators](#) may be used as the first character if followed by a non-[space](#) “safe” character, as this causes no ambiguity.

```

( ns-char - c-indicator )
[126] ns-plain-first(c) ::= | ( ( “?” | “:” | “-” )
/* Followed by an ns-plain-safe\(c\) */ )

```

Plain scalars must never contain the “:” and “#” character combinations. Such combinations would cause ambiguity with [mapping key: value pairs](#) and [comments](#). In addition, inside [flow collections](#), or when used as [implicit keys](#), plain scalars must not contain the “[”, “]”, “{”, “}” and “.” characters. These characters would cause ambiguity with [flow collection](#) structures.

```

c = flow-out ⇒ ns-plain-safe-out
c = flow-in  ⇒ ns-plain-safe-in
[127] ns-plain-safe(c) ::= c = block-key ⇒ ns-plain-safe-out
c = flow-key ⇒ ns-plain-safe-in
[128] ns-plain-safe-out ::= ns-char
[129] ns-plain-safe-in  ::= ns-char - c-flow-indicator

```

```

[130] ns-plain-char(c) ::= ( ns-plain-safe\(c\) - “.” - “#” )
| ( /* An ns-char preceding */ “#” )
| ( “.” /* Followed by an ns-plain-safe\(c\) */ )

```

Example 7.10. Plain Characters

```

                                %YAML 1.2
                                ---
# Outside flow collection:      !!seq [
- ::vector                     !!str "::vector",
- ": - ()"                     !!str ": - ()",
- Up, up, and away!           !!str "Up, up, and away!",
- -123                         !!int "-123",
- http://example.com/foo#bar   !!str "http://example.com/foo#bar",
# Inside flow collection:      !!seq [
- [ ::vector,                  !!str "::vector",
  ": - ()",                     !!str ": - ()",
  "Up, up and away!",           !!str "Up, up, and away!",
  -123,                         !!int "-123",
  http://example.com/foo#bar ]  !!str "http://example.com/foo#bar"
                                ],
                                ]

```

Legend:

[ns-plain-first\(c\)](#) Not ns-plain-first(c) [ns-plain-char\(c\)](#) Not ns-pl

Plain scalars are further restricted to a single line when contained inside an [implicit key](#).

```

                                c = flow-out ⇒ ns-plain-multi-line\(n,c\)
                                c = flow-in  ⇒ ns-plain-multi-line\(n,c\)
[131] ns-plain(n,c) ::= c = block-key ⇒ ns-plain-one-line\(c\)
                                c = flow-key ⇒ ns-plain-one-line\(c\)
[132] nb-ns-plain-in-line(c) ::= ( s-white* ns-plain-char\(c\) )*
[133] ns-plain-one-line(c) ::= ns-plain-first\(c\) nb-ns-plain-in-line\(c\)

```

Example 7.11. Plain Implicit Keys

```

implicit block key : [
  implicit flow key : value,
]
Legend:
  ns-plain-one-line\(c\)
%YAML 1.2
---
!!map {
  ? !!str "implicit block key"
  : !!seq [
    !!map {
      ? !!str "implicit flow key"
      : !!str "value",
    }
  ]
}

```

All leading and trailing [white space](#) characters are excluded from the [content](#). Each continuation line must therefore contain at least one non-[space](#) character. Empty lines, if any, are consumed as part of the [line folding](#).

- [134] $s\text{-}ns\text{-}plain\text{-}next\text{-}line(n,c) ::= \text{[s-flow-folded\(n\)](#) [ns-plain-char\(c\)](#) [nb-ns-plain-in-line\(c\)](#)}$
- [135] $ns\text{-}plain\text{-}multi\text{-}line(n,c) ::= \text{[ns-plain-one-line\(c\)](#) [s-ns-plain-next-line\(n,c\)^*](#)}$

Example 7.12. Plain Lines

```

1st non-empty↓
↓
·2nd non-empty·
→3rd non-empty
%YAML 1.2
---
!!str "1st non-empty\n\
      2nd non-empty \
      3rd non-empty"

```

Legend:
[nb-ns-plain-in-line\(c\)](#) [s-ns-plain-next-line\(n,c\)](#)

7.4. Flow Collection Styles

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 an [implicit key](#) ([flow-key context](#) or [block-key context](#)). Flow collection entries are terminated by the “,” *indicator*. The final “,” may be omitted. This does not cause ambiguity because flow collection entries can never be [completely empty](#).

$$[136] \quad \text{in-flow}(c) ::= \begin{array}{l} c = \text{flow-out} \Rightarrow \text{flow-in} \\ c = \text{flow-in} \Rightarrow \text{flow-in} \\ c = \text{block-key} \Rightarrow \text{flow-key} \\ c = \text{flow-key} \Rightarrow \text{flow-key} \end{array}$$

7.4.1. Flow Sequences

Flow sequence content is denoted by surrounding “[” and “]” characters.

[137] $\text{c-flow-sequence}(n,c) ::= \text{"[" s-separate}(n,c)\text{?}$
 $\text{ns-s-flow-seq-entries}(n,\text{in-flow}(c))\text{? "]"}$

Sequence entries are separated by a “+” character.

[138] $\text{ns-s-flow-seq-entries}(n,c) ::= (\text{“,” } \text{ns-s-flow-seq-entries}(n,c) \text{“” } \text{ns-flow-seq-entry}(n,c) \text{ s-separate}(n,c) \text{“” } \text{ns-s-flow-seq-entries}(n,c) \text{“”})?$

Example 7.13. Flow Sequence

```

%YAML 1.2
---
!!seq [
  !!seq [
    !!str "one",
    !!str "two",
  ],
  !!seq [
    !!str "three",
    !!str "four",
  ]
]

```

Legend:

- c-sequence-start c-sequence-end
- ns-flow-seq-entry(n,c)

],

Any [flow node](#) may be used as a flow sequence entry. In addition, YAML provides a [compact notation](#) for the case where a flow sequence entry is a [mapping](#) with a [single key: value pair](#).

[139] $\text{ns-flow-seq-entry}(n,c) ::= \text{ns-flow-pair}(n,c) \mid \text{ns-flow-node}(n,c)$

Example 7.14. Flow Sequence Entries

	%YAML 1.2
[---
"double	!!seq [
quoted", 'single	!!str "double quoted",
quoted',	!!str "single quoted",
plain	!!str "plain text",
text, [nested],	!!seq [
single: pair,	!!str "nested",
]],
	!!map {
Legend:	? !!str "single"
ns-flow-node(n,c)	: !!str "pair",
ns-flow-pair(n,c)	},
]

7.4.2. Flow Mappings

Flow mappings are denoted by surrounding “{” and “}” characters.

[140] $\text{c-flow-mapping}(n,c) ::= \text{“}\{” \text{ s-separate}(n,c)?$
 $\text{ns-s-flow-map-entries}(n,\text{in-flow}(c))? \text{“}\}”$

Mapping entries are separated by a [“,”](#) character.

[141] $\text{ns-s-flow-map-entries}(n,c) ::= (\text{“}\text{,”} \text{ s-separate}(n,c)?$
 $\text{ns-flow-map-entry}(n,c) \text{ s-separate}(n,c)?$

[ns-s-flow-map-entries\(n,c\)? \)?](#)

Example 7.15. Flow Mappings

```
%YAML 1.2
---
!!seq [
- { one : two , three: four , }
- {five: six,seven : eight}
Legend:
  c-mapping-start c-mapping-end
  ns-flow-map-entry\(n,c\)
  !!map {
    ? !!str "one" : !!str "two",
    ? !!str "three" : !!str "four",
  },
  !!map {
    ? !!str "five" : !!str "six",
    ? !!str "seven" : !!str "eight",
  },
]
```

If the optional “?” *mapping key indicator* is specified, the rest of the entry may be [completely empty](#).

```
[142]      ns-flow-map-entry(n,c) ::= ( “?” s-separate\(n,c\)
                                         ns-flow-map-explicit-entry\(n,c\)
                                         | ns-flow-map-implicit-entry\(n,c\)
                                         | ns-flow-map-implicit-entry\(n,c\)
[143]      ns-flow-map-explicit-
          entry(n,c) ::= | ( e-node /* Key */
                           e-node /* Value */ )
```

Example 7.16. Flow Mapping Entries

```
{
? explicit: entry,
implicit: entry,
?°°
}
Legend:
  ns-flow-map-explicit-entry\(n,c\)
  ns-flow-map-implicit-entry\(n,c\)
  e-node
%YAML 1.2
---
!!map {
  ? !!str "explicit" : !!str "entry"
  ? !!str "implicit" : !!str "entry"
  ? !!null "" : !!null "",
}
```

Normally, YAML insists the “:” *mapping value indicator* be [separated](#) from the [value](#) by [white space](#). A benefit of this restriction is that the “:” character can be used inside [plain scalars](#), as long as it is not followed by [white space](#). This allows for unquoted URLs and timestamps. It is also a potential source for confusion as “a:1” is a [plain scalar](#) and not a [key: value pair](#).

Note that the [value](#) may be [completely empty](#) since its existence is indicated by the “:”.

```

[144] ns-flow-map-implicit-entry(n,c) ::= ns-flow-map-yaml-key-
                                          entry\(n,c\)
                                          | c-ns-flow-map-empty-key-
                                          entry\(n,c\)
                                          | c-ns-flow-map-json-key-
                                          entry\(n,c\)
                                          | ns-flow-yaml-node\(n,c\)
[145] ns-flow-map-yaml-key-      ::= ( ( s-separate\(n,c\)?
      entry(n,c)                | c-ns-flow-map-separate-
                               value\(n,c\) )
                               | e-node )
[146] c-ns-flow-map-empty-key-  ::= e-node /* Key */
      entry(n,c)                | c-ns-flow-map-separate-
                               value\(n,c\)
                               | “:” /* Not followed by an
[147] c-ns-flow-map-separate-  ::= ( ( s-separate\(n,c\) ns-flow-
      value(n,c)                node\(n,c\) )
                               | e-node /* Value */ )

```

Example 7.17. Flow Mapping Separate Values

```

{
unquoted:::"separate",           %YAML 1.2
http://foo.com,                 ---
omitted value:°,                !!map {
°:·omitted key,                  ? !!str "unquoted" : !!str "sepa

```

```

}                                ? !!str "http://foo.com" : !!nul
                                ? !!str "omitted value" : !!null
Legend:                          ? !!null "" : !!str "omitted key
  ns-flow-yaml-node\(n,c\) e-node }
  c-ns-flow-map-separate-value\(n,c\)

```

To ensure [JSON compatibility](#), if a [key](#) inside a flow mapping is [JSON-like](#), YAML allows the following [value](#) to be specified adjacent to the “:”. This causes no ambiguity, as all [JSON-like keys](#) are surrounded by [indicators](#). However, as this greatly reduces readability, YAML [processors](#) should [separate](#) the [value](#) from the “:” on output, even in this case.

```

[148]      c-ns-flow-map-json-key-      c-flow-json-node\(n,c\)
              entry(n,c) ::= ( ( s-separate\(n,c\)?
                                c-ns-flow-map-adjacent-
                                value\(n,c\) )
                                | e-node )

[149]      c-ns-flow-map-adjacent-      “:” ( ( s-separate\(n,c\)?
              value(n,c) ::=      ns-flow-node\(n,c\) )
                                | e-node ) /* Value */

```

Example 7.18. Flow Mapping Adjacent Values

```

{                                %YAML 1.2
"adjacent":value,                ---
"readable":·value,              !!map {
"empty":°                        ? !!str "adjacent" : !!str "valu
}                                ? !!str "readable" : !!str "valu
Legend:                          ? !!str "empty" : !!null "",
  c-flow-json-node\(n,c\) e-node }
  c-ns-flow-map-adjacent-value\(n,c\)

```

A more compact notation is usable inside [flow sequences](#), if the [mapping](#) contains a *single key: value pair*. This notation does not require the surrounding “{” and “}” characters. Note that it is not possible to specify any [node properties](#) for the [mapping](#) in this case.

Example 7.19. Single Pair Flow Mappings

```
[                                %YAML 1.2
foo: bar                        ---
]                                !!seq [
                                !!map { ? !!str "foo" : !!str "bar" }
Legend:                          ]
  ns-flow-pair\(n,c\)
```

If the “?” indicator is explicitly specified, [parsing](#) is unambiguous, and the syntax is identical to the general case.

```
[150]    ns-flow-pair(n,c) ::= ( “?” s-separate\(n,c\)
                                ns-flow-map-explicit-entry\(n,c\) )
                                | ns-flow-pair-entry\(n,c\)
```

Example 7.20. Single Pair Explicit Entry

```
[                                %YAML 1.2
? foo                          ---
  bar : baz                    !!seq [
                                !!map {
                                ? !!str "foo bar"
                                : !!str "baz",
Legend:                          },
  ns-flow-map-explicit-entry\(n,c\) ]
```

If the “?” indicator is omitted, [parsing](#) needs to see past the *implicit key* to recognize it as such. To limit the amount of lookahead required, the “:” indicator must appear at most 1024 Unicode characters beyond the start of the [key](#). In addition, the [key](#) is restricted to a single line.

Note that YAML allows arbitrary [nodes](#) to be used as [keys](#). In particular, a [key](#) may be a [sequence](#) or a [mapping](#). Thus, without the above restrictions, practical one-pass [parsing](#) would have been impossible to implement.

[ns-flow-pair-yaml-key-entry\(n,c\)](#)

- [151] ns-flow-pair-entry(n,c) ::= | [c-ns-flow-map-empty-key-entry\(n,c\)](#)
| [c-ns-flow-pair-json-key-entry\(n,c\)](#)
- [152] ns-flow-pair-yaml-key-entry(n,c) ::= [ns-s-implicit-yaml-key\(flow-key\)](#)
[c-ns-flow-map-separate-value\(n,c\)](#)
- [153] c-ns-flow-pair-json-key-entry(n,c) ::= [c-s-implicit-json-key\(flow-key\)](#)
[c-ns-flow-map-adjacent-value\(n,c\)](#)
[ns-flow-yaml-node\(n/a,c\)](#) s-separate-in-line?
- [154] ns-s-implicit-yaml-key(c) ::= /* At most 1024 characters altogether */
[c-flow-json-node\(n/a,c\)](#) s-separate-in-line?
- [155] c-s-implicit-json-key(c) ::= /* At most 1024 characters altogether */

Example 7.21. Single Pair Implicit Entries

```

%YAML 1.2
---
!!seq [
  !!seq [
    !!map {
      ? !!str "YAML"
      : !!str "separate"
    },
  ],
  !!seq [
    !!map {
      ? !!null ""
      : !!str "empty key entry"
    },
  ],
  !!seq [
    !!map {
      ? !!map {
        ? !!str "JSON"
        : !!str "like"
      } : "adjacent",
    },
  ],
]

```

- [YAML.: separate]

- [°: empty key entry]

- [{JSON: like}:adjacent]

Legend:

[ns-s-implicit-yaml-key](#)

[c-s-implicit-json-key](#)

[e-node](#) Value

Example 7.22. Invalid Implicit Keys

```
[ foo                                ERROR:
  bar: invalid,                      - The foo bar key spans mult
  "foo...>1K characters...bar": invalid ]- The foo...bar key is too l
```

7.5. Flow Nodes

JSON-like [flow styles](#) all have explicit start and end [indicators](#). The only [flow style](#) that does not have this property is the [plain scalar](#). Note that none of the “JSON-like” styles is actually acceptable by JSON. Even the [double-quoted style](#) is a superset of the JSON string format.

- [156] `ns-flow-yaml-content(n,c) ::= ns-plain\(n,c\)`
- [157] `c-flow-json-content(n,c) ::= c-flow-sequence\(n,c\) | c-flow-mapping\(n,c\) | c-single-quoted\(n,c\) | c-double-quoted\(n,c\)`
- [158] `ns-flow-content(n,c) ::= ns-flow-yaml-content\(n,c\) | c-flow-json-content\(n,c\)`

Example 7.23. Flow Content

```
- [ a, b ]           %YAML 1.2
- { a: b }           ---
- "a"                !!seq [
- 'b'                 !!seq [ !!str "a", !!str "b" ],
- c                   !!map { ? !!str "a" : !!str "b" },
                      !!str "a",
                      !!str "b",
                      !!str "c",
Legend:
c-flow-json-content\(n,c\)
ns-flow-yaml-content\(n,c\) ]
```

A complete [flow node](#) also has optional [node properties](#), except for [alias nodes](#) which refer to the [anchored node properties](#).

- [159] `ns-flow-yaml-node(n,c) ::= c-ns-alias-node | ns-flow-yaml-content\(n,c\) | (c-ns-properties\(n,c\) (s-separate\(n,c\) ns-flow-yaml-content\(n,c\)) | e-scalar))`

- [160] $\text{c-flow-json-node}(n,c) ::= (\text{c-ns-properties}(n,c) \text{ s-separate}(n,c))?$
 $\text{c-flow-json-content}(n,c)$
 c-ns-alias-node
 $| \text{ns-flow-content}(n,c)$
- [161] $\text{ns-flow-node}(n,c) ::= (\text{c-ns-properties}(n,c)$
 $((\text{s-separate}(n,c)$
 $\text{ns-flow-content}(n,c))$
 $| \text{e-scalar}))$

Example 7.24. Flow Nodes

```
- !!str "a"           %YAML 1.2
- 'b'               ---
- &anchor "c"        !!seq [
- *anchor            !!str "a",
- !!str°             !!str "b",
                        &A !!str "c",
                        *A,
                        !!str "",
Legend:
  c-flow-json-node\(n,c\)
  ns-flow-yaml-node\(n,c\) ]
```

Chapter 8. Block Styles

YAML's *block styles* employ [indentation](#) rather than [indicators](#) to denote structure. This results in a more human readable (though less compact) notation.

8.1. Block Scalar Styles

YAML provides two *block scalar styles*, [literal](#) and [folded](#). Each provides a different trade-off between readability and expressive power.

8.1.1. Block Scalar Headers

[Block scalars](#) are controlled by a few [indicators](#) given in a *header* preceding the [content](#) itself. This header is followed by a non-content [line break](#) with an optional [comment](#). This is the only case where a [comment](#) must not be followed by additional [comment](#) lines.

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

Example 8.1. Block Scalar Header

```
- | # Empty header↓      %YAML 1.2  
  literal               ---  
- >1 # Indentation indicator↓ !!seq [  
  .folded               !!str "literal\n",  
- |+ # Chomping indicator↓  !!str ".folded\n",  
  keep                  !!str "keep\n\n",  
                        !!str ".strip",  
- >1- # Both indicators↓  ]  
  .strip  
                        Legend:  
                        c-b-block-header\(m,t\)
```

8.1.1.1. Block Indentation Indicator

Typically, the [indentation](#) level of a [block scalar](#) is detected from its first non-[empty](#) line. It is an error for any of the leading [empty lines](#) to contain more

[spaces](#) than the first non-[empty line](#).

Detection fails when the first non-[empty line](#) contains leading content [space](#) characters. [Content](#) may safely start with a [tab](#) or a “#” character.

When detection would fail, 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](#), relative to its parent [node](#).

It is always valid to specify an indentation indicator for a [block scalar](#) node, though a YAML [processor](#) should only emit an explicit indentation indicator for cases where detection will fail.

[163] c-indentation-indicator(m) ::= [ns-dec-digit](#) ⇒ m = [ns-dec-digit](#) - #x30
/* Empty */ ⇒ m = auto-detect()

Example 8.2. Block Indentation Indicator

```
- |°                                     %YAML 1.2
·detected                             ---
- >°                                  !!seq [
·                                     !!str "detected\n",
·                                     !!str "\n\n# detected\n",
··                                  !!str "·explicit\n",
··# detected                        !!str "\t·detected\n",
- |1                                 ]
·explicit
- >°
·→
·detected
```

Legend:
[c-indentation-indicator\(m\)](#)
[s-indent\(n\)](#)

Example 8.3. Invalid Block Scalar Indentation Indicators

```
- |
·
·text
- >
·text
·text
```

ERROR:

- A leading all-space line must not have too many spaces.
- A following text line must not be less indented.

- |2
- text
- The text is less indented than the indicated level.

8.1.1.2. Block Chomping Indicator

Chomping controls how final [line breaks](#) and trailing [empty lines](#) are interpreted. YAML provides three chomping methods:

Strip

Stripping is specified by the “-” *chomping indicator*. In this case, the final [line break](#) and any trailing [empty lines](#) are excluded from the [scalar’s content](#).

Clip

Clipping is the default behavior used if no explicit chomping indicator is specified. In this case, the final [line break](#) character is preserved in the [scalar’s content](#). However, any trailing [empty lines](#) are excluded from the [scalar’s content](#).

Keep

Keeping is specified by the “+” *chomping indicator*. In this case, the final [line break](#) and any trailing [empty lines](#) are considered to be part of the [scalar’s content](#). These additional lines are not subject to [folding](#).

The chomping method used is a [presentation detail](#) and must not be used to convey [content](#) information.

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

The interpretation of the final [line break](#) of a [block scalar](#) is controlled by the chomping indicator specified in the [block scalar header](#).

```
[165]  b-chomped-last(t) ::= t = strip ⇒ b-non-content | /* End of file */
                                     t = clip ⇒ b-as-line-feed | /* End of file */
                                     t = keep ⇒ b-as-line-feed | /* End of file */
```

Example 8.4. Chomping Final Line Break

```
strip: |-                                %YAML 1.2
  text↓                                ---
clip: |                                  !!map {
  text↓                                ? !!str "strip"
  : !!str "text",
keep: |+                                ? !!str "clip"
  text↓                                : !!str "text\n",
Legend:                                ? !!str "keep"
  b-non-content b-as-line-feed      : !!str "text\n",
}
```

The interpretation of the trailing [empty lines](#) following a [block scalar](#) is also controlled by the chomping indicator specified in the [block scalar header](#).

```
[166]      l-chomped-empty(n,t) ::= t = strip ⇒ l-strip-empty\(n\)
                                     t = clip  ⇒ l-strip-empty\(n\)
                                     t = keep  ⇒ l-keep-empty\(n\)
[167]      l-strip-empty(n) ::= ( s-indent\(<=n\) b-non-content ) *
                                     l-trail-comments\(n\)?
[168]      l-keep-empty(n) ::= l-empty\(n,block-in\) *
                                     l-trail-comments\(n\)?
```

Explicit [comment](#) lines may follow the trailing [empty lines](#). 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. This is the only case where the [indentation](#) of [comment](#) lines is constrained.

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

Example 8.5. Chomping Trailing Lines

```
# Strip
# Comments:
strip: |-                                %YAML 1.2
```


Example 8.7. Literal Scalar

```
|↓
·literal↓
·→text↓                                %YAML 1.2
↓                                      ---
                                      !!str "literal\n\ttext\n"

Legend:
  c-l+literal\(n\)
```

Inside literal scalars, all ([indented](#)) characters are considered to be [content](#), including [white space](#) characters. Note that all [line break](#) characters are [normalized](#). In addition, [empty lines](#) are not [folded](#), though final [line breaks](#) and trailing [empty lines](#) are [chomped](#).

There is no way to escape characters inside literal scalars. This restricts them to [printable](#) characters. In addition, there is no way to break a long literal line.

```
[171]  l-nb-literal-text(n) ::= l-empty\(n,block-in\)\*
                                       s-indent\(n\) nb-char+
[172]  b-nb-literal-next(n) ::= b-as-line-feed
                                       l-nb-literal-text\(n\)
                                       ( l-nb-literal-text\(n\) b-nb-literal-next\(n\)\*
[173]  l-literal-content(n,t) ::= b-chomped-last\(t\) )?
                                       l-chomped-empty\(n,t\)
```

Example 8.8. Literal Content

```
|                                %YAML 1.2
·                                ---
··                               !!str "\n\nliteral\n·\n\n\ttext\n"
··literal↓
···↓
··
··text↓
↓
·# Comment
```

Legend:

- [l-nb-literal-text\(n\)](#)
- [b-nb-literal-next\(n\)](#)
- [b-chomped-last\(t\)](#)
- [l-chomped-empty\(n,t\)](#)

8.1.3. Folded Style

The *folded style* is denoted by the “>” *indicator*. It is similar to the [literal style](#); however, folded scalars are subject to [line folding](#).

[174] $c-l+folded(n) ::= \text{“>” } c-b-block-header(m,t) \text{ } l-folded-content(n+m,t)$

Example 8.9. Folded Scalar

```
>↓
·folded↓
·text↓
↓
%YAML 1.2
---
!!str "folded text\n"

Legend:
c-l+folded(n)
```

[Folding](#) allows long lines to be broken anywhere a single [space](#) character separates two non-[space](#) characters.

[175] $s-nb-folded-text(n) ::= s-indent(n) \text{ } ns-char \text{ } nb-char^*$

[176] $l-nb-folded-lines(n) ::= s-nb-folded-text(n) \text{ } (\text{ } b-l-folded(n,block-in) \text{ } s-nb-folded-text(n))^*$

Example 8.10. Folded Lines

```
>
·folded↓
·line↓
↓
·next
·line↓
  * bullet
    * list
    * lines
·last↓

%YAML 1.2
---
!!str "\n\
      folded line\n\
      next line\n\
      \  * bullet\n\
      \  * list\n\
      \  * lines\n\
      last line\n"

Legend:
```

```

·line↓                                s-nb-folded-text\(n\)
                                       l-nb-folded-lines\(n\)
# Comment

```

(The following three examples duplicate this example, each highlighting different productions.)

Lines starting with [white space](#) characters (*more-indented* lines) are not [folded](#).

```

[177]    s-nb-spaced-text(n) ::= s-indent\(n\) s-white nb-char*
[178]          b-l-spaced(n) ::= b-as-line-feed
                               l-empty\(n,block-in\)*
[179]    l-nb-spaced-lines(n) ::= s-nb-spaced-text\(n\)
                                   ( b-l-spaced\(n\) s-nb-spaced-text\(n\) )*

```

Example 8.11. More Indented Lines

```

>
folded                                %YAML 1.2
line                                ---
                                   !!str "\n\
next                                folded line\n\
line                                next line\n\
                                   \ * bullet\n
... * bullet↓                       \n\
↓                                   \ * list\n\
... * list↓                         \ * lines\n\
... * lines↓                       \n\
                                   last line\n"

last
line
# Comment

```

Legend:

[s-nb-spaced-text\(n\)](#)
[l-nb-spaced-lines\(n\)](#)

[Line breaks](#) and [empty lines](#) separating folded and more-indented lines are also not [folded](#).

```

[180] l-nb-same-lines(n) ::= l-empty\(n,block-in\)*

```

(l-nb-folded-lines(n) | l-nb-spaced-lines(n))
l-nb-same-lines(n)

Example 8.12. Empty Separation Lines

```
>
↓
folded
line↓
↓
next
line↓
  * bullet
    * list
    * line↓
↓
last
line

# Comment
```

```
%YAML 1.2
---
!!str "\n\
      folded line\n\
      next line\n\
      \ * bullet\n\
      \n\
      \ * list\n\
      \ * lines\n\
      \n\
      last line\n"
```

Legend:

[b-as-line-feed](#)
(separation) [l-empty\(n,c\)](#)

The final [line break](#), and trailing [empty lines](#) if any, are subject to [chomping](#) and are never [folded](#).

[182] $\text{l-folded-content}(n,t) ::= (\text{l-nb-diff-lines}(n) \text{ b-chomped-last}(t))?$
 $\text{l-chomped-empty}(n,t)$

Example 8.13. Final Empty Lines

```
> folded                                %YAML 1.2
line                                  ---
                                     !!str "\n\
next                                folded line\n\
line                               next line\n\
    * bullet                       \  * bullet\n
                                     \n\
                                     \  * list\n
```

```
* list
* line

last
line↓
↓
# Comment
```

```
\ * lines\n\
\n\
last line\n"
```

Legend:

[b-chomped-last\(t\)](#) [l-chomped-empty\(n,](#)

8.2. Block Collection Styles

For readability, *block collections styles* are not denoted by any [indicator](#). Instead, YAML uses a lookahead method, where a block collection is distinguished from a [plain scalar](#) only when a [key: value pair](#) or a [sequence entry](#) is seen.

8.2.1. Block Sequences

A *block sequence* is simply a series of [nodes](#), each denoted by a leading “-” *indicator*. The “-” indicator must be [separated](#) from the [node](#) by [white space](#). This allows “-” to be used as the first character in a [plain scalar](#) if followed by a non-space character (e.g. “-1”).

[183] l+block-sequence(n) ::= ([s-indent\(n+m\)](#) [c-l-block-seq-entry\(n+m\)](#))+
/* For some fixed auto-detected m > 0 */

[184] c-l-block-seq-entry(n) ::= [“-”](#) /* Not followed by an [ns-char](#) */
[s-l+block-indented\(n,block-in\)](#)

Example 8.14. Block Sequence

```
%YAML 1.2
---
block sequence:
  - one
  - two : three
Legend:
  c-l-block-seq-entry\(n\)
  auto-detected s-indent\(n\)
```

```
!!map {
  ? !!str "block sequence"
  : !!seq [
    !!str "one",
    !!map {
      ? !!str "two"
      : !!str "three"
    },
  ],
}
```

The entry [node](#) may be either [completely empty](#), be a nested [block node](#), or use a *compact in-line notation*. The compact notation may be used when the entry is itself a nested [block collection](#). In this case, both the “-” indicator and the following [spaces](#) are considered to be part of the [indentation](#) of the nested

[collection](#). Note that it is not possible to specify [node properties](#) for such a [collection](#).

```

[185] s-l+block-indented(n,c) ::= ( s-indent\(m\)
                                     ( ns-l-compact-sequence\(n+1+m\)
                                     | ns-l-compact-mapping\(n+1+m\) ) )
                                     | s-l+block-node\(n,c\)
                                     | ( e-node s-l-comments )

[186] ns-l-compact-sequence(n) ::= c-l-block-seq-entry\(n\)
                                     ( s-indent\(n\) c-l-block-seq-entry\(n\) )*
```

Example 8.15. Block Sequence Entry Types

	%YAML 1.2
- ° # Empty	---
-	!!seq [
block node	!!null "",
- ·· one # Compact	!!str "block node\n",
- ·· two # sequence	!!seq [
- one: two # Compact mapping	!!str "one"
	!!str "two",
Legend:],
Empty	!!map {
s-l+block-node(n,c)	? !!str "one"
ns-l-compact-sequence(n)	: !!str "two",
ns-l-compact-mapping(n)	},
]

8.2.2. Block Mappings

A *Block mapping* is a series of entries, each [presenting](#) a [key: value pair](#).

```

[187] l+block-mapping(n) ::= ( s-indent\(n+m\) ns-l-block-map-entry\(n+m\) )+
/* For some fixed auto-detected m > 0 */
```

Example 8.16. Block Mappings

%YAML 1.2

```

block mapping:      ---
·key: value↓        !!map {
                    ? !!str "block mapping"
Legend:             : !!map {
  ns-l-block-map-entry(n)  ? !!str "key"
  auto-detected s-indent(n) : !!str "value",
                        },
                    }

```

If the “?” indicator is specified, the optional value node must be specified on a separate line, denoted by the “:.” indicator. Note that YAML allows here the same [compact in-line notation](#) described above for [block sequence](#) entries.

```

[188] ns-l-block-map-entry(n) ::= c-l-block-map-explicit-entry\(n\)
                                     | ns-l-block-map-implicit-entry\(n\)
[189] c-l-block-map-explicit-entry\(n\) ::= ( c-l-block-map-explicit-key\(n\)
                                     | l-block-map-explicit-value\(n\)
                                     | e-node )
[190] c-l-block-map-explicit-key(n) ::= “?” s-l+block-indented\(n,block-out\)
[191] l-block-map-explicit-value(n) ::= s-indent\(n\)
                                     | “:.” s-l+block-indented\(n,block-out\)

```

Example 8.17. Explicit Block Mapping Entries

```

? explicit key # Empty value↓ ° %YAML 1.2
? |
  block key↓                    ---
  :.- one # Explicit compact    !!map {
  :.- two # block value↓        ? !!str "explicit key"
                                : !!str "",
Legend:                         ? !!str "block key\n"
  c-l-block-map-explicit-key\(n\) : !!seq [
  l-block-map-explicit-value\(n\) !!str "one",
  e-node                        !!str "two",
                                ],
                                }

```

If the “?” indicator is omitted, [parsing](#) needs to see past the [implicit key](#), in the

same way as in the [single key: value pair flow mapping](#). Hence, such [keys](#) are subject to the same restrictions; they are limited to a single line and must not span more than 1024 Unicode characters.

```
[192]      ns-l-block-map-implicit-  ( ns-s-block-map-implicit-key
          entry(n)                  := | e-node )
                                     | c-l-block-map-implicit-value\(n\)
                                     | c-s-implicit-json-key\(block-key\)
[193]      ns-s-block-map-implicit-key := | ns-s-implicit-yaml-key\(block-
                                           key\)
```

In this case, the [value](#) may be specified on the same line as the [implicit key](#). Note however that in block mappings the [value](#) must never be adjacent to the “:”, as this greatly reduces readability and is not required for [JSON compatibility](#) (unlike the case in [flow mappings](#)).

There is no compact notation for in-line [values](#). Also, while both the [implicit key](#) and the [value](#) following it may be empty, the “:” indicator is mandatory. This prevents a potential ambiguity with multi-line [plain scalars](#).

```
[194] c-l-block-map-implicit-value(n) ::= “:” ( s-l+block-node\(n,block-out\)
                                                | ( e-node s-l-comments ) )
```

Example 8.18. Implicit Block Mapping Entries

```
plain key: in-line value      %YAML 1.2
°:° # Both empty             ---
"quoted key":                !!map {
- entry                       ? !!str "plain key"
                              : !!str "in-line value",
                              ? !!null ""
Legend:                       : !!null "",
                              ? !!str "quoted key"
  ns-s-block-map-implicit-key ? !!seq [ !!str "entry" ],
  c-l-block-map-implicit-value\(n\) :
                              }
```

A [compact in-line notation](#) is also available. This compact notation may be nested inside [block sequences](#) and explicit block mapping entries. Note that it is

not possible to specify [node properties](#) for such a nested mapping.

[195] $\text{ns-l-compact-mapping}(n) ::= \text{ns-l-block-map-entry}(n) \text{ (} \text{s-indent}(n) \text{ ns-l-block-map-entry}(n) \text{)}^*$

Example 8.19. Compact Block Mappings

```

- sun: yellow↓
- ? earth: blue↓
  : moon: white↓
Legend:
  ns-l-compact-mapping(n)

%YAML 1.2
---
!!seq [
  !!map {
    !!str "sun" : !!str "yellow",
  },
  !!map {
    ? !!map {
      ? !!str "earth"
      : !!str "blue"
    },
    : !!map {
      ? !!str "moon"
      : !!str "white"
    },
  },
]

```

8.2.3. Block Nodes

YAML allows [flow nodes](#) to be embedded inside [block collections](#) (but not vice-versa). [Flow nodes](#) must be [indented](#) by at least one more [space](#) than the parent [block collection](#). Note that [flow nodes](#) may begin on a following line.

It is at this point that [parsing](#) needs to distinguish between a [plain scalar](#) and an [implicit key](#) starting a nested [block mapping](#).

[196] $\text{s-l+block-node}(n,c) ::= \text{s-l+block-in-block}(n,c) \mid \text{s-l+flow-in-block}(n)$

[197] $\text{s-l+flow-in-block}(n) ::= \text{s-separate}(n+1,\text{flow-out}) \mid \text{ns-flow-node}(n+1,\text{flow-out}) \text{ s-l-comments}$

Example 8.20. Block Node Types

```

-↓                                     %YAML 1.2
.."flow in block"↓                   ---
-·>                                !!seq [
  Block scalar↓                      !!str "flow in block",
-·!!map # Block collection          !!str "Block scalar\n",
  foo : bar↓                        !!map {
                                   ? !!str "foo"
                                   : !!str "bar",
Legend:
  s-l+flow-in-block\(n\)           },
  s-l+block-in-block\(n,c\) ]

```

The block [node's properties](#) may span across several lines. In this case, they must be [indented](#) by at least one more [space](#) than the [block collection](#), regardless of the [indentation](#) of the [block collection](#) entries.

```

[198]      s-l+block-in- ::= s-l+block-scalar\(n,c\) | s-l+block-
              block(n,c)    collection\(n,c\)
                           s-separate\(n+1,c\)
[199] s-l+block-scalar(n,c) ::= ( c-ns-properties\(n+1,c\) s-separate\(n+1,c\) )?
                                ( c-l+literal\(n\) | c-l+folded\(n\) )

```

Example 8.21. Block Scalar Nodes

```

literal: |2                          %YAML 1.2
..value                              ---
folded:↓                             !!map {
...!foo                             ? !!str "literal"
..>1                                : !!str "value",
..value                             ? !!str "folded"
                                   : !<!foo> "value",
Legend:
  c-l+literal\(n\) c-l+folded\(n\) }

```

Since people perceive the “-” indicator as [indentation](#), nested [block sequences](#) may be [indented](#) by one less [space](#) to compensate, except, of course, if nested inside another [block sequence](#) ([block-out context](#) vs. [block-in context](#)).

Chapter 9. YAML Character Stream

9.1. Documents

A YAML character [stream](#) may contain several *documents*. Each document is completely independent from the rest.

9.1.1. Document Prefix

A document may be preceded by a *prefix* specifying the [character encoding](#), and optional [comment](#) lines. Note that all [documents](#) in a stream must use the same [character encoding](#). However it is valid to re-specify the [encoding](#) using a [byte order mark](#) for each [document](#) in the stream. This makes it easier to concatenate streams.

The existence of the optional prefix does not necessarily indicate the existence of an actual [document](#).

[202] l-document-prefix ::= [c-byte-order-mark](#)? [l-comment](#)*

Example 9.1. Document Prefix

```
↔# Comment
# lines           %YAML 1.2
Document         ---
Legend:          !!str "Document"
l-document-prefix
```

9.1.2. Document Markers

Using [directives](#) creates a potential ambiguity. It is valid to have a “%” character at the start of a line (e.g. as the first character of the second line of a [plain scalar](#)). How, then, to distinguish between an actual [directive](#) and a [content](#) line that happens to start with a “%” character?

The solution is the use of two special *marker* lines to control the processing of [directives](#), one at the start of a [document](#) and one at the end.

At the start of a [document](#), lines beginning with a “[%](#)” character are assumed to be [directives](#). The (possibly empty) list of [directives](#) is terminated by a *directives end marker* line. Lines following this marker can safely use “[%](#)” as the first character.

At the end of a [document](#), a *document end marker* line is used to signal the [parser](#) to begin scanning for [directives](#) again.

The existence of this optional *document suffix* does not necessarily indicate the existence of an actual following [document](#).

Obviously, the actual [content](#) lines are therefore forbidden to begin with either of these markers.

```
[203]      c-directives-end ::= “_” “_” “_”
[204]      c-document-end  ::= “.” “.” “.”
[205]      l-document-suffix ::= c-document-end s-l-comments
                               /* Start of line */
[206]      c-forbidden    ::= ( c-directives-end | c-document-end )
                               ( b-char | s-white | /* End of file */ )
```

Example 9.2. Document Markers

```
%YAML 1.2
---
Document
... # Suffix

%YAML 1.2
---
!!str "Document"

Legend:
c-directives-end c-document-end
l-document-suffix
```

9.1.3. Bare Documents

A *bare document* does not begin with any [directives](#) or [marker](#) lines. Such documents are very “clean” as they contain nothing other than the [content](#). In this case, the first non-comment line may not start with a “[%](#)” first character.

Document [nodes](#) are [indented](#) as if they have a parent [indented](#) at -1 [spaces](#). Since a [node](#) must be more [indented](#) than its parent [node](#), this allows the document's [node](#) to be [indented](#) at zero or more [spaces](#).

```
[207]      l-bare-document ::= s-l+block-node\(-1,block-in\)
                               /* Excluding c-forbidden content */
```

Example 9.3. Bare Documents

Bare	%YAML 1.2
document	---
...	!!str "Bare document"
# No document	%YAML 1.2
...	---
	!!str "%!PS-Adobe-2.0\n"
%!PS-Adobe-2.0 # Not the first line	Legend:
	l-bare-document

9.1.4. Explicit Documents

An *explicit document* begins with an explicit [directives end marker](#) line but no [directives](#). Since the existence of the [document](#) is indicated by this [marker](#), the [document](#) itself may be [completely empty](#).

```
[208]      l-explicit-document ::= (c-directives-end
                                   (l-bare-document
                                   | (e-node s-l-comments ) )
```

Example 9.4. Explicit Documents

---	%YAML 1.2
{ matches	---
% : 20 }	!!map {
...	!!str "matches %": !!int "20"
---	}
# Empty	...
...	%YAML 1.2

Legend:

[l-explicit-document](#)

!!null ""

9.1.5. Directives Documents

A *directives document* begins with some [directives](#) followed by an explicit [directives end marker](#) line.

[209] l-directive-document ::= [l-directive](#)⁺
[l-explicit-document](#)

Example 9.5. Directives Documents

```
%YAML 1.2
--- |
%!PS-Adobe-2.0          %YAML 1.2
...                      ---
%YAML1.2                 !!str "%!PS-Adobe-2.0\n"
---                      ...
# Empty                 %YAML 1.2
...                      ---
                        !!null ""
```

Legend:

[l-explicit-document](#)

9.2. Streams

A YAML *stream* consists of zero or more [documents](#). Subsequent [documents](#) require some sort of separation [marker](#) line. If a [document](#) is not terminated by a [document end marker](#) line, then the following [document](#) must begin with a [directives end marker](#) line.

The stream format is intentionally “sloppy” to better support common use cases, such as stream concatenation.

```
[210]      l-any-      <a href="#">l-directive-document</a>
document ::= | <a href="#">l-explicit-document</a>
              | <a href="#">l-bare-document</a>
              <a href="#">l-document-prefix</a>* <a href="#">l-any-document</a>?
[211] l-yaml-stream ::= ( <a href="#">l-document-suffix</a>+ <a href="#">l-document-prefix</a>* <a href="#">l-any-
                        <a href="#">document</a>?
                        | <a href="#">l-document-prefix</a>* <a href="#">l-explicit-document</a>? )*
```

Example 9.6. Stream

Document	%YAML 1.2
---	---
# Empty	!!str "Document"
...	...
%YAML 1.2	%YAML 1.2
---	---
matches %: 20	!!null ""
	...
Legend:	%YAML 1.2
l-any-document	---
l-document-suffix	!!map {
l-explicit-document	!!str "matches %": !!int "20"
	}

A sequence of bytes is a *well-formed stream* if, taken as a whole, it complies with the above **l-yaml-stream** production.

Some common use case that can take advantage of the YAML stream structure are:

Appending to Streams

Allowing multiple [documents](#) in a single stream makes YAML suitable for log files and similar [applications](#). Note that each [document](#) is independent of the rest, allowing for heterogeneous log file entries.

Concatenating Streams

Concatenating two YAML streams requires both to use the same [character encoding](#). In addition, it is necessary to separate the last [document](#) of the first stream and the first [document](#) of the second stream. This is easily ensured by inserting a [document end marker](#) between the two streams. Note that this is safe regardless of the content of either stream. In particular, either or both may be empty, and the first stream may or may not already contain such a marker.

Communication Streams

The [document end marker](#) allows signaling the end of a [document](#) without closing the stream or starting the next [document](#). This allows the receiver to complete processing a [document](#) without having to wait for the next one to arrive. The sender may also transmit "keep-alive" messages in the form of [comment](#) lines or repeated [document end markers](#) without signalling the start of the next [document](#).

Chapter 10. Recommended Schemas

A YAML *schema* is a combination of a set of [tags](#) and a mechanism for [resolving non-specific tags](#).

10.1. Failsafe Schema

The *failsafe schema* is guaranteed to work with any YAML [document](#). It is therefore the recommended [schema](#) for generic YAML tools. A YAML [processor](#) should therefore support this [schema](#), at least as an option.

10.1.1. Tags

10.1.1.1. Generic Mapping

URI:

`tag:yaml.org,2002:map`

Kind:

[Mapping](#).

Definition:

[Represents](#) an associative container, where each [key](#) is unique in the association and mapped to exactly one [value](#). YAML places no restrictions on the type of [keys](#); in particular, they are not restricted to being [scalars](#). Example [bindings](#) to [native](#) types include Perl's hash, Python's dictionary, and Java's Hashtable.

Example 10.1. !!map Examples

```
Block style: !!map
  Clark : Evans
  Ingy  : döt Net
  Oren   : Ben-Kiki
```

```
Flow style: !!map { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }
```

10.1.1.2. Generic Sequence

URI:

tag:yaml.org,2002:seq

Kind:

[Sequence](#).

Definition:

[Represents](#) a collection indexed by sequential integers starting with zero. Example [bindings](#) to [native](#) types include Perl's array, Python's list or tuple, and Java's array or Vector.

Example 10.2. !!seq Examples

Block style: !!seq

- Clark Evans
- Ingy döt Net
- Oren Ben-Kiki

Flow style: !!seq [Clark Evans, Ingy döt Net, Oren Ben-Kiki]

10.1.1.3. Generic String

URI:

tag:yaml.org,2002:str

Kind:

[Scalar](#).

Definition:

[Represents](#) a Unicode string, a sequence of zero or more Unicode characters. This type is usually [bound](#) to the [native](#) language's string type, or, for languages lacking one (such as C), to a character array.

Canonical Form:

The obvious.

Example 10.3. `!!str` Examples

```
Block style: !!str |-  
  String: just a theory.
```

```
Flow style: !!str "String: just a theory."
```

10.1.2. Tag Resolution

All [nodes](#) with the [“!” non-specific tag](#) are [resolved](#), by the standard [convention](#), to `“tag:yaml.org,2002:seq”`, `“tag:yaml.org,2002:map”`, or `“tag:yaml.org,2002:str”`, according to their [kind](#).

All [nodes](#) with the [“?” non-specific tag](#) are left [unresolved](#). This constrains the [application](#) to deal with a [partial representation](#).

10.2. JSON Schema

The *JSON schema* is the lowest common denominator of most modern computer languages, and allows [parsing](#) JSON files. A YAML [processor](#) should therefore support this [schema](#), at least as an option. It is also strongly recommended that other [schemas](#) should be based on it.

10.2.1. Tags

The JSON [schema](#) uses the following [tags](#) in addition to those defined by the [failsafe](#) schema:

10.2.1.1. Null

URI:

tag:yaml.org,2002:null

Kind:

[Scalar](#).

Definition:

[Represents](#) the lack of a value. This is typically [bound](#) to a [native](#) null-like value (e.g., **undef** in Perl, **None** in Python). Note that a null is different from an empty string. Also, a [mapping](#) entry with some [key](#) and a null [value](#) is valid, and different from not having that [key](#) in the [mapping](#).

Canonical Form:

null.

Example 10.4. !!null Examples

```
!!null null: value for null key
key with null value: !!null null
```

10.2.1.2. Boolean

URI:

tag:yaml.org,2002:bool

Kind:

[Scalar](#).

Definition:

[Represents](#) a true/false value. In languages without a [native](#) Boolean type (such as C), is usually [bound](#) to a native integer type, using one for true and zero for false.

Canonical Form:

Either **true** or **false**.

Example 10.5. !!bool Examples

```
YAML is a superset of JSON: !!bool true
Pluto is a planet: !!bool false
```

10.2.1.3. Integer

URI:

tag:yaml.org,2002:int

Kind:

[Scalar](#).

Definition:

[Represents](#) arbitrary sized finite mathematical integers. Scalars of this type should be [bound](#) to a [native](#) integer data type, if possible.

Some languages (such as Perl) provide only a “number” type that allows for both integer and floating-point values. A YAML [processor](#) may use such a type for integers, as long as they round-trip properly.

In some languages (such as C), an integer may overflow the [native](#) type’s storage capability. A YAML [processor](#) may reject such a value as an error, truncate it with a warning, or find some other manner to round-trip it. In general, integers representable using 32 binary digits should safely round-trip through most systems.

Canonical Form:

Decimal integer notation, with a leading “-” character for negative values, matching the regular expression `0 | -? [1-9] [0-9]*`

Example 10.6. !!int Examples

```
negative: !!int -12
zero: !!int 0
positive: !!int 34
```

10.2.1.4. Floating Point

URI:

tag:yaml.org,2002:float

Kind:

[Scalar](#).

Definition:

[Represents](#) an approximation to real numbers, including three special values (positive and negative infinity, and “not a number”).

Some languages (such as Perl) provide only a “number” type that allows for both integer and floating-point values. A YAML [processor](#) may use such a type for floating-point numbers, as long as they round-trip properly.

Not all floating-point values can be stored exactly in any given [native](#) type. Hence a float value may change by “a small amount” when round-tripped. The supported range and accuracy depends on the implementation, though 32 bit IEEE floats should be safe. Since YAML does not specify a particular accuracy, using floating-point [mapping keys](#) requires great care and is not recommended.

Canonical Form:

Either `0`, `.inf`, `-.inf`, `.nan`, or scientific notation matching the regular expression `-? [1-9] (\. [0-9]* [1-9])? (e [-+] [1-9] [0-9]*)?`.

Example 10.7. `!!float` Examples

```
negative: !!float -1
zero: !!float 0
positive: !!float 2.3e4
infinity: !!float .inf
not a number: !!float .nan
```

10.2.2. Tag Resolution

The [JSON schema tag resolution](#) is an extension of the [failsafe schema tag resolution](#).

All [nodes](#) with the “[!](#)” [non-specific tag](#) are [resolved](#), by the standard [convention](#), to “`tag:yaml.org,2002:seq`”, “`tag:yaml.org,2002:map`”, or “`tag:yaml.org,2002:str`”, according to their [kind](#).

[Collections](#) with the “[?](#)” [non-specific tag](#) (that is, [untagged collections](#)) are [resolved](#) to “`tag:yaml.org,2002:seq`” or “`tag:yaml.org,2002:map`” according to their [kind](#).

[Scalars](#) with the “?” [non-specific tag](#) (that is, [plain scalars](#)) are matched with a list of regular expressions (first match wins, e.g. `0` is resolved as `!!int`). In principle, JSON files should not contain any [scalars](#) that do not match at least one of these. Hence the YAML [processor](#) should consider them to be an error.

<i>Regular expression</i>	<i>Resolved to tag</i>
<code>null</code>	<code>tag:yaml.org,2002:null</code>
<code>true false</code>	<code>tag:yaml.org,2002:bool</code>
<code>-? (0 [1-9] [0-9]*)</code>	<code>tag:yaml.org,2002:int</code>
<code>-? (0 [1-9] [0-9]*) (\. [0-9]*)? ([eE] [-+]? [0-9]+)?</code>	<code>tag:yaml.org,2002:float</code>
<code>*</code>	Error

Example 10.8. JSON Tag Resolution

<pre> A null: null Booleans: [true, false] Integers: [0, -0, 3, -19] Floats: [0., -0.0, 12e03, -2E+05] Invalid: [True, Null, 0o7, 0x3A, +12.3] </pre>	<pre> %YAML 1.2 --- !!map { !!str "A null" : !!null !!str "Booleans: !!seq [!!bool "true", !!bool], !!str "Integers": !!seq !!int "0", !!int "-0", !!int "3", !!int "-19"], !!str "Floats": !!seq [!!float "0.", !!float !!float "12e03", !!flo], !!str "Invalid": !!seq [# Rejected by the sche True, Null, 0o7, 0x3A,], } ... </pre>
---	---

10.3. Core Schema

The *Core schema* is an extension of the [JSON schema](#), allowing for more human-readable [presentation](#) of the same types. This is the recommended default [schema](#) that YAML [processor](#) should use unless instructed otherwise. It is also strongly recommended that other [schemas](#) should be based on it.

10.3.1. Tags

The core [schema](#) uses the same [tags](#) as the [JSON schema](#).

10.3.2. Tag Resolution

The [core schema tag resolution](#) is an extension of the [JSON schema tag resolution](#).

All [nodes](#) with the “[!](#)” [non-specific tag](#) are [resolved](#), by the standard [convention](#), to “`tag:yaml.org,2002:seq`”, “`tag:yaml.org,2002:map`”, or “`tag:yaml.org,2002:str`”, according to their [kind](#).

[Collections](#) with the “[?](#)” [non-specific tag](#) (that is, [untagged collections](#)) are [resolved](#) to “`tag:yaml.org,2002:seq`” or “`tag:yaml.org,2002:map`” according to their [kind](#).

[Scalars](#) with the “[?](#)” [non-specific tag](#) (that is, [plain scalars](#)) are matched with an extended list of regular expressions. However, in this case, if none of the regular expressions matches, the [scalar](#) is [resolved](#) to `tag:yaml.org,2002:str` (that is, considered to be a string).

Regular expression

`null | Null | NULL | ~`

`/* Empty */`

`true | True | TRUE | false | False | FALSE`

`[-+]? [0-9]+`

`0o [0-7]+`

`0x [0-9a-fA-F]+`

Resolved to tag

`tag:yaml.org,2002:null`

`tag:yaml.org,2002:null`

`tag:yaml.org,2002:bool`

`tag:yaml.org,2002:int (Base 10)`

`tag:yaml.org,2002:int (Base 8)`

`tag:yaml.org,2002:int (Base 16)`


```

[-+]? ( \. [0-9]+ | [0-9]+ ( \. [0-9]* )? ) ( [eE] [-+]? [0-9]+ )?
[-+]? ( \.inf | \.Inf | \.INF )
\.nan | \.NaN | \.NAN
*

```

```

tag:yaml.org,2002:float (
tag:yaml.org,2002:float (
tag:yaml.org,2002:float (
tag:yaml.org,2002:str (D

```

Example 10.9. Core Tag Resolution

```

A null: null
Also a null: # Empty
Not a null: ""
Booleans: [ true, True, false, FALSE ]
Integers: [ 0, 0o7, 0x3A, -19 ]
Floats: [ 0., -0.0, .5, +12e03, -2E+05 ]
Also floats: [ .inf, -.Inf, +.INF, .NAN ]

```

```

%YAML 1.2
---
!!map {
  !!str "A null" : !!null
  !!str "Also a null" : !!
  !!str "Not a null" : !!s
  !!str "Booleans: !!seq [
    !!bool "true", !!bool
    !!bool "false", !!bool
  ],
  !!str "Integers": !!seq
    !!int "0", !!int "0o7"
    !!int "0x3A", !!int "-
  ],
  !!str "Floats": !!seq [
    !!float "0.", !!float
    !!float "+12e03", !!fl
  ],
  !!str "Also floats": !!s
    !!float ".inf", !!floa
    !!float "+.INF", !!flo
  ],
}
...

```

10.4. Other Schemas

None of the above recommended [schemas](#) preclude the use of arbitrary explicit [tags](#). Hence YAML [processors](#) for a particular programming language typically provide some form of [local tags](#) that map directly to the language's [native data structures](#) (e.g., `!ruby/object:Set`).

While such [local tags](#) are useful for ad-hoc [applications](#), they do not suffice for stable, interoperable cross-[application](#) or cross-platform data exchange.

Interoperable [schemas](#) make use of [global tags](#) (URIs) that [represent](#) the same data across different programming languages. In addition, an interoperable [schema](#) may provide additional [tag resolution](#) rules. Such rules may provide additional regular expressions, as well as consider the path to the [node](#). This allows interoperable [schemas](#) to use [untagged nodes](#).

It is strongly recommended that such [schemas](#) be based on the [core schema](#) defined above. In addition, it is strongly recommended that such [schemas](#) make as much use as possible of the the *YAML tag repository* at <http://yaml.org/type/>. This repository provides recommended [global tags](#) for increasing the portability of YAML [documents](#) between different [applications](#).

The tag repository is intentionally left out of the scope of this specification. This allows it to evolve to better support YAML [applications](#). Hence, developers are encouraged to submit new “universal” types to the repository. The yaml-core mailing list at <http://lists.sourceforge.net/lists/listinfo/yaml-core> is the preferred method for such submissions, as well as raising any questions regarding this draft.

Index

Indicators

! tag indicator, [Tags](#), [Indicator Characters](#), [Node Tags](#)

! local tag, [Tags](#), [Tag Handles](#), [Tag Prefixes](#), [Node Tags](#)

! non-specific tag, [Resolved Tags](#), [Node Tags](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#)

! primary tag handle, [Tag Handles](#)

!! secondary tag handle, [Tag Handles](#)

!...! named handle, [Tag Handles](#), [Node Tags](#)

" double-quoted style, [Indicator Characters](#), [Double-Quoted Style](#)

comment, [Collections](#), [Indicator Characters](#), [Comments](#), [Plain Style](#), [Block Indentation Indicator](#)

% directive, [Indicator Characters](#), [Directives](#), [Document Markers](#), [Bare Documents](#)

% escaping in URI, [Tags](#), [Miscellaneous Characters](#), [Node Tags](#)

& anchor, [Structures](#), [Indicator Characters](#), [Node Anchors](#)

' reserved indicator, [Indicator Characters](#)

' single-quoted style, [Indicator Characters](#), [Single-Quoted Style](#)

* alias, [Structures](#), [Indicator Characters](#), [Alias Nodes](#)

+ keep chomping, [Block Chomping Indicator](#)

, end flow entry, [Collections](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#), [Plain Style](#), [Flow Collection Styles](#), [Flow Sequences](#), [Flow Mappings](#)

- block sequence entry, [Introduction](#), [Collections](#), [Structures](#), [Production Parameters](#), [Indicator Characters](#), [Indentation Spaces](#), [Plain Style](#), [Block Collection Styles](#), [Block Sequences](#), [Block Nodes](#)

- strip chomping, [Block Chomping Indicator](#)

: mapping value, [Introduction](#), [Collections](#), [Structures](#), [Indicator Characters](#), [Indentation Spaces](#), [Plain Style](#), [Flow Mappings](#), [Block Mappings](#)

<...> verbatim tag, [Node Tags](#)

> folded style, [Scalars](#), [Indicator Characters](#), [Folded Style](#)

? mapping key, [Structures](#), [Indicator Characters](#), [Indentation Spaces](#), [Plain Style](#), [Flow Mappings](#), [Block Mappings](#)

? non-specific tag, [Resolved Tags](#), [Node Tags](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#)

@ reserved indicator, [Indicator Characters](#)

[start flow sequence, [Collections](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#), [Plain Style](#), [Flow Sequences](#)

\ escaping in double-quoted scalars, [Escaped Characters](#), [Double-Quoted Style](#)

] end flow sequence, [Collections](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#), [Plain Style](#), [Flow Sequences](#)

{ start flow mapping, [Collections](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#), [Plain Style](#), [Flow Mappings](#)

| literal style, [Scalars](#), [Indicator Characters](#), [Literal Style](#)

} end flow mapping, [Collections](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#), [Plain Style](#), [Flow Mappings](#)

prefix, [Document Prefix](#)

A

alias, [Introduction](#), [Prior Art](#), [Structures](#), [Dump](#), [Serialization Tree](#), [Anchors and Aliases](#), [Loading Failure Points](#), [Well-Formed Streams and Identified Aliases](#), [Resolved Tags](#), [Indicator Characters](#), [Node Anchors](#), [Flow Styles](#), [Alias Nodes](#), [Flow Nodes](#)

identified, [Structures](#), [Anchors and Aliases](#), [Well-Formed Streams and Identified Aliases](#)

unidentified, [Loading Failure Points](#), [Well-Formed Streams and Identified Aliases](#)

anchor, [Structures](#), [Dump](#), [Serialization Tree](#), [Anchors and Aliases](#), [Well-Formed Streams and Identified Aliases](#), [Resolved Tags](#), [Indicator Characters](#), [Node Properties](#), [Flow Styles](#), [Alias Nodes](#), [Flow Nodes](#)

application, [Introduction](#), [Prior Art](#), [Tags](#), [Processing YAML Information](#), [Dump](#), [Information Models](#), [Tags](#), [Serialization Tree](#), [Keys Order](#), [Resolved Tags](#), [Available Tags](#), [Tag Prefixes](#), [Node Tags](#), [Streams](#), [Tag Resolution](#), [Other Schemas](#)

B

block scalar header, [Comments](#), [Block Scalar Headers](#), [Block Chomping](#)

[Indicator](#)

byte order mark, [Character Encodings](#), [Document Prefix](#)

C

character encoding, [Character Encodings](#), [Document Prefix](#), [Streams](#)

in URI, [Miscellaneous Characters](#)

chomping, [Production Parameters](#), [Line Folding](#), [Block Chomping Indicator](#), [Literal Style](#), [Folded Style](#)

clip, [Production Parameters](#), [Block Chomping Indicator](#)

keep, [Production Parameters](#), [Block Chomping Indicator](#)

strip, [Production Parameters](#), [Block Chomping Indicator](#)

collection, [Prior Art](#), [Representation Graph](#), [Nodes](#), [Node Comparison](#), [Anchors and Aliases](#), [Node Styles](#), [Comments](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Tag Resolution](#), [Tag Resolution](#)

comment, [Collections](#), [Processes](#), [Dump](#), [Load](#), [Presentation Stream](#), [Comments](#), [Resolved Tags](#), [Indicator Characters](#), [Comments](#), [Separation Lines](#), [Plain Style](#), [Block Scalar Headers](#), [Block Chomping Indicator](#), [Document Prefix](#), [Streams](#)

compose, [Processes](#), [Load](#), [Keys Order](#), [Anchors and Aliases](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Available Tags](#), [Node Anchors](#)

construct, [Processes](#), [Load](#), [Serialization Tree](#), [Loading Failure Points](#), [Recognized and Valid Tags](#), [Available Tags](#), [Flow Styles](#), [Generic Mapping](#), [Generic Sequence](#), [Generic String](#), [Null](#), [Boolean](#), [Integer](#)

content, [Structures](#), [Dump](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Loading Failure Points](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Character Encodings](#), [Line Break Characters](#), [Escaped Characters](#), [Indentation Spaces](#), [Separation Spaces](#), [Line Prefixes](#), [Empty Lines](#), [Line Folding](#), [Comments](#), [Directives](#), [Tag Handles](#), [Node Properties](#), [Node Tags](#), [Node Anchors](#), [Alias Nodes](#), [Empty Nodes](#), [Flow Scalar Styles](#), [Double-Quoted Style](#), [Single-Quoted Style](#), [Plain Style](#), [Block Scalar Headers](#), [Block Indentation Indicator](#), [Block Chomping Indicator](#), [Literal Style](#), [Document Markers](#), [Bare Documents](#)

valid, [Recognized and Valid Tags](#)

context, [Production Parameters](#), [Plain Style](#)

block-in, [Production Parameters](#), [Block Nodes](#)
block-key, [Production Parameters](#), [Flow Collection Styles](#)
block-out, [Production Parameters](#), [Block Nodes](#)
flow-in, [Production Parameters](#), [Flow Collection Styles](#)
flow-key, [Production Parameters](#), [Flow Collection Styles](#)
flow-out, [Production Parameters](#), [Flow Collection Styles](#)

D

directive, [Structures](#), [Dump](#), [Load](#), [Presentation Stream](#), [Directives](#), [Indicator Characters](#), [Directives](#), [Document Markers](#), [Bare Documents](#), [Explicit Documents](#), [Directives Documents](#)

reserved, [Directives](#), [Directives](#)
TAG, [Tags](#), [Directives](#), [Indicator Characters](#), [Directives](#), [“TAG” Directives](#), [Node Tags](#)
YAML, [Directives](#), [Directives](#), [“YAML” Directives](#)

document, [Prior Art](#), [Structures](#), [Presentation Stream](#), [Directives](#), [Loading Failure Points](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Character Encodings](#), [Line Break Characters](#), [“YAML” Directives](#), [Tag Prefixes](#), [Alias Nodes](#), [Documents](#), [Document Prefix](#), [Document Markers](#), [Explicit Documents](#), [Streams](#), [Failsafe Schema](#), [Other Schemas](#)

bare, [Bare Documents](#)
directives, [Directives Documents](#)
explicit, [Explicit Documents](#)
suffix, [Document Markers](#)

dump, [Processes](#), [Dump](#)

E

empty line, [Prior Art](#), [Scalars](#), [Empty Lines](#), [Line Folding](#), [Block Indentation Indicator](#), [Block Chomping Indicator](#), [Literal Style](#), [Folded Style](#)
equality, [Relation to JSON](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Scalar Formats](#), [Loading Failure Points](#), [Recognized and Valid Tags](#)
escaping

in double-quoted scalars, [Prior Art](#), [Scalars](#), [Character Set](#), [Character Encodings](#), [Miscellaneous Characters](#), [Escaped Characters](#), [Double-Quoted Style](#)
in single-quoted scalars, [Single-Quoted Style](#)
in URIs, [Miscellaneous Characters](#)
non-content line break, [Double-Quoted Style](#)

I

identity, [Node Comparison](#)
indicator, [Introduction](#), [Prior Art](#), [Collections](#), [Node Styles](#), [Production Parameters](#), [Indicator Characters](#), [Line Folding](#), [Plain Style](#), [Flow Mappings](#), [Flow Nodes](#), [Block Styles](#), [Block Scalar Headers](#), [Block Collection Styles](#)

indentation, [Block Indentation Indicator](#)
reserved, [Indicator Characters](#)

information model, [Information Models](#)
invalid content, [Loading Failure Points](#), [Recognized and Valid Tags](#)

J

JSON compatibility, [Character Set](#), [Character Encodings](#), [Line Break Characters](#), [Escaped Characters](#), [Comments](#), [“YAML” Directives](#), [Flow Mappings](#), [Block Mappings](#)
JSON-like, [Flow Mappings](#), [Flow Nodes](#)

K

key, [Relation to JSON](#), [Structures](#), [Dump](#), [Information Models](#), [Representation Graph](#), [Nodes](#), [Node Comparison](#), [Serialization Tree](#), [Keys Order](#), [Resolved Tags](#), [Indicator Characters](#), [Flow Mappings](#), [Block Mappings](#), [Generic Mapping](#), [Null](#), [Floating Point](#)

implicit, [Separation Lines](#), [Double-Quoted Style](#), [Single-Quoted Style](#), [Plain Style](#), [Flow Collection Styles](#), [Flow Mappings](#), [Block Mappings](#), [Block Nodes](#)
order, [Processes](#), [Dump](#), [Load](#), [Information Models](#), [Serialization Tree](#), [Keys Order](#)

key: value pair, [Introduction](#), [Collections](#), [Structures](#), [Nodes](#), [Keys Order](#), [Node Styles](#), [Plain Style](#), [Flow Mappings](#), [Block Collection Styles](#), [Block Mappings](#) kind, [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Node Styles](#), [Resolved Tags](#), [Node Tags](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#)

L

line break, [Prior Art](#), [Scalars](#), [Production Parameters](#), [Production Naming Conventions](#), [Line Break Characters](#), [White Space Characters](#), [Empty Lines](#), [Line Folding](#), [Comments](#), [Double-Quoted Style](#), [Block Scalar Headers](#), [Block Chomping Indicator](#), [Literal Style](#), [Folded Style](#)

non-ASCII, [Line Break Characters](#), [“YAML” Directives](#)
normalization, [Line Break Characters](#), [Literal Style](#)

line folding, [Prior Art](#), [Scalars](#), [Line Folding](#), [Flow Styles](#), [Double-Quoted Style](#), [Single-Quoted Style](#), [Plain Style](#), [Block Chomping Indicator](#), [Folded Style](#)

block, [Line Folding](#), [Folded Style](#)
flow, [Line Folding](#), [Double-Quoted Style](#)

line prefix, [Line Prefixes](#), [Empty Lines](#)
load, [Processes](#), [Load](#), [Loading Failure Points](#)

failure point, [Load](#), [Loading Failure Points](#)

M

mapping, [Introduction](#), [Prior Art](#), [Relation to JSON](#), [Collections](#), [Structures](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Keys Order](#), [Resolved Tags](#), [Plain Style](#), [Flow Sequences](#), [Flow Mappings](#), [Generic Mapping](#), [Null](#)

marker, [Presentation Stream](#), [Document Markers](#), [Bare Documents](#), [Explicit Documents](#), [Streams](#)

directives end, [Structures](#), [Document Markers](#), [Explicit Documents](#), [Directives Documents](#), [Streams](#)
document end, [Structures](#), [Document Markers](#), [Streams](#)

more-indented, [Scalars](#), [Line Folding](#), [Folded Style](#)

N

native data structure, [Introduction](#), [Goals](#), [Prior Art](#), [Relation to JSON](#), [Processing YAML Information](#), [Processes](#), [Dump](#), [Load](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Serialization Tree](#), [Loading Failure Points](#), [Recognized and Valid Tags](#), [Available Tags](#), [Node Tags](#), [Flow Styles](#), [Generic Mapping](#), [Generic Sequence](#), [Generic String](#), [Null](#), [Boolean](#), [Integer](#), [Floating Point](#), [Other Schemas](#)

need not, [Terminology](#)

node, [Structures](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Serialization Tree](#), [Keys Order](#), [Anchors and Aliases](#), [Presentation Stream](#), [Node Styles](#), [Comments](#), [Loading Failure Points](#), [Well-Formed Streams and Identified Aliases](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Syntax Conventions](#), [Indentation Spaces](#), [Node Properties](#), [Node Tags](#), [Node Anchors](#), [Alias Nodes](#), [Empty Nodes](#), [Flow Mappings](#), [Flow Nodes](#), [Block Indentation Indicator](#), [Block Sequences](#), [Bare Documents](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#), [Other Schemas](#)

completely empty, [Empty Nodes](#), [Flow Collection Styles](#), [Flow Mappings](#), [Block Sequences](#), [Explicit Documents](#)

property, [Node Properties](#), [Alias Nodes](#), [Empty Nodes](#), [Flow Mappings](#), [Flow Nodes](#), [Block Sequences](#), [Block Mappings](#), [Block Nodes](#)

root, [Representation Graph](#), [Resolved Tags](#)

P

parse, [Load](#), [Presentation Stream](#), [Resolved Tags](#), [Production Parameters](#), [Line Break Characters](#), [Escaped Characters](#), [Tag Handles](#), [Node Tags](#), [Flow Mappings](#), [Block Mappings](#), [Block Nodes](#), [Document Markers](#), [JSON Schema](#)

present, [Processing YAML Information](#), [Dump](#), [Load](#), [Nodes](#), [Node Comparison](#), [Presentation Stream](#), [Scalar Formats](#), [Character Set](#), [Miscellaneous Characters](#), [Node Tags](#), [Alias Nodes](#), [Block Mappings](#), [Core Schema](#)

presentation, [Processing YAML Information](#), [Information Models](#), [Presentation Stream](#), [Production Parameters](#)

detail, [Dump](#), [Load](#), [Information Models](#), [Presentation Stream](#), [Node Styles](#), [Scalar Formats](#), [Comments](#), [Directives](#), [Resolved Tags](#), [Character](#)

[Encodings](#), [Line Break Characters](#), [Escaped Characters](#), [Indentation Spaces](#), [Separation Spaces](#), [Line Prefixes](#), [Line Folding](#), [Comments](#), [Directives](#), [Tag Handles](#), [Node Tags](#), [Flow Scalar Styles](#), [Block Chomping Indicator](#)

printable character, [Introduction](#), [Prior Art](#), [Character Set](#), [White Space Characters](#), [Escaped Characters](#), [Single-Quoted Style](#), [Literal Style](#)
processor, [Terminology](#), [Processing YAML Information](#), [Dump](#), [Node Comparison](#), [Presentation Stream](#), [Directives](#), [Well-Formed Streams and Identified Aliases](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Available Tags](#), [Character Set](#), [Character Encodings](#), [Line Break Characters](#), [Miscellaneous Characters](#), [Comments](#), [Directives](#), [“YAML” Directives](#), [Tag Handles](#), [Node Tags](#), [Node Anchors](#), [Flow Mappings](#), [Block Indentation Indicator](#), [Failsafe Schema](#), [JSON Schema](#), [Integer](#), [Floating Point](#), [Tag Resolution](#), [Core Schema](#), [Other Schemas](#)

R

represent, [Introduction](#), [Prior Art](#), [Dump](#), [Tags](#), [Node Comparison](#), [Keys Order](#), [Generic Mapping](#), [Generic Sequence](#), [Generic String](#), [Null](#), [Boolean](#), [Integer](#), [Floating Point](#), [Other Schemas](#)

representation, [Processing YAML Information](#), [Processes](#), [Dump](#), [Load](#), [Information Models](#), [Representation Graph](#), [Nodes](#), [Node Comparison](#), [Serialization Tree](#), [Keys Order](#), [Anchors and Aliases](#), [Presentation Stream](#), [Node Styles](#), [Scalar Formats](#), [Comments](#), [Directives](#), [Available Tags](#), [Node Anchors](#)

complete, [Loading Failure Points](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Available Tags](#)

partial, [Loading Failure Points](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Tag Resolution](#)

required, [Terminology](#)

S

scalar, [Introduction](#), [Prior Art](#), [Scalars](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Scalar Formats](#), [Comments](#), [Recognized and Valid Tags](#), [Line Break Characters](#), [Separation Spaces](#), [Line Prefixes](#), [Comments](#), [Empty Nodes](#), [Block Chomping Indicator](#), [Generic Mapping](#), [Generic String](#), [Null](#), [Boolean](#), [Integer](#), [Floating Point](#), [Tag Resolution](#), [Tag Resolution](#)

canonical form, [Prior Art](#), [Tags](#), [Node Comparison](#), [Scalar Formats](#), [Loading Failure Points](#)

content format, [Dump](#), [Load](#), [Tags](#), [Node Comparison](#), [Presentation Stream](#), [Scalar Formats](#), [Loading Failure Points](#)

schema, [Recommended Schemas](#), [Failsafe Schema](#), [JSON Schema](#), [Tags](#), [Core Schema](#), [Tags](#), [Other Schemas](#)

core, [Core Schema](#), [Tag Resolution](#), [Other Schemas](#)

failsafe, [Tags](#), [Failsafe Schema](#), [Tags](#), [Tag Resolution](#)

JSON, [Tags](#), [JSON Schema](#), [Tag Resolution](#), [Core Schema](#), [Tags](#), [Tag Resolution](#)

sequence, [Introduction](#), [Prior Art](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Keys Order](#), [Resolved Tags](#), [Flow Mappings](#), [Generic Sequence](#)

serialization, [Processing YAML Information](#), [Processes](#), [Dump](#), [Load](#), [Information Models](#), [Serialization Tree](#), [Anchors and Aliases](#), [Presentation Stream](#), [Node Styles](#), [Scalar Formats](#), [Comments](#), [Directives](#), [Node Anchors](#)

detail, [Dump](#), [Load](#), [Information Models](#), [Keys Order](#), [Anchors and Aliases](#), [Node Anchors](#)

serialize, [Introduction](#), [Prior Art](#), [Relation to JSON](#), [Dump](#), [Load](#), [Keys Order](#), [Anchors and Aliases](#), [Alias Nodes](#)

shall, [Terminology](#)

space, [Prior Art](#), [Scalars](#), [White Space Characters](#), [Indentation Spaces](#), [Line Folding](#), [Single-Quoted Style](#), [Plain Style](#), [Block Indentation Indicator](#), [Folded Style](#), [Block Sequences](#), [Block Nodes](#), [Bare Documents](#)

indentation, [Introduction](#), [Prior Art](#), [Collections](#), [Dump](#), [Load](#), [Information Models](#), [Node Styles](#), [Resolved Tags](#), [Production Parameters](#), [Production Naming Conventions](#), [Indentation Spaces](#), [Separation Spaces](#), [Line Prefixes](#), [Line Folding](#), [Comments](#), [Separation Lines](#), [Directives](#), [Block Styles](#), [Block Indentation Indicator](#), [Block Chomping Indicator](#), [Literal Style](#), [Block Sequences](#), [Block Nodes](#), [Bare Documents](#)

separation, [Separation Spaces](#), [Comments](#), [Flow Mappings](#), [Block Sequences](#)

white, [Production Naming Conventions](#), [White Space Characters](#),

[Separation Spaces](#), [Line Prefixes](#), [Line Folding](#), [Comments](#), [Double-Quoted Style](#), [Single-Quoted Style](#), [Plain Style](#), [Flow Mappings](#), [Literal Style](#), [Folded Style](#), [Block Sequences](#)

[stream](#), [Prior Art](#), [Processing YAML Information](#), [Processes](#), [Dump](#), [Load](#), [Presentation Stream](#), [Loading Failure Points](#), [Well-Formed Streams and Identified Aliases](#), [Resolved Tags](#), [Syntax Conventions](#), [Character Set](#), [Character Encodings](#), [Miscellaneous Characters](#), [Comments](#), [Tag Prefixes](#), [Empty Nodes](#), [Documents](#), [Streams](#)

[ill-formed](#), [Load](#), [Loading Failure Points](#), [Well-Formed Streams and Identified Aliases](#)

[well-formed](#), [Well-Formed Streams and Identified Aliases](#), [Streams](#)

[style](#), [Dump](#), [Load](#), [Information Models](#), [Presentation Stream](#), [Node Styles](#), [Scalar Formats](#), [Resolved Tags](#), [Node Tags](#), [Double-Quoted Style](#), [Plain Style](#)

[block](#), [Prior Art](#), [Scalars](#), [Node Styles](#), [Production Parameters](#), [Indentation Spaces](#), [Block Styles](#), [Block Sequences](#)

[collection](#), [Collections](#), [Structures](#), [Indentation Spaces](#), [Flow Collection Styles](#), [Block Collection Styles](#), [Block Sequences](#), [Block Nodes](#)

[folded](#), [Scalars](#), [Node Styles](#), [Indicator Characters](#), [Line Folding](#), [Block Scalar Styles](#), [Literal Style](#), [Folded Style](#)

[literal](#), [Prior Art](#), [Scalars](#), [Node Styles](#), [Indicator Characters](#), [Block Scalar Styles](#), [Literal Style](#), [Folded Style](#)

[mapping](#), [Node Styles](#), [Production Parameters](#), [Block Mappings](#), [Block Nodes](#)

[scalar](#), [Node Styles](#), [Block Scalar Styles](#), [Block Scalar Headers](#), [Block Indentation Indicator](#), [Block Chomping Indicator](#)

[sequence](#), [Collections](#), [Node Styles](#), [Production Parameters](#), [Indicator Characters](#), [Block Sequences](#), [Block Mappings](#), [Block Nodes](#)

[compact block collection](#), [Node Styles](#), [Block Sequences](#), [Block Mappings](#)
[flow](#), [Prior Art](#), [Collections](#), [Scalars](#), [Node Styles](#), [Production Parameters](#), [Line Folding](#), [Flow Styles](#), [Flow Sequences](#), [Flow Nodes](#), [Block Nodes](#)

[collection](#), [Syntax Conventions](#), [Production Parameters](#), [Indicator Characters](#), [Miscellaneous Characters](#), [Node Tags](#), [Node Anchors](#),

[Plain Style](#), [Flow Collection Styles](#)

double-quoted, [Prior Art](#), [Scalars](#), [Node Styles](#), [Syntax Conventions](#), [Character Set](#), [Character Encodings](#), [Indicator Characters](#), [Escaped Characters](#), [Flow Scalar Styles](#), [Double-Quoted Style](#), [Flow Nodes](#) mapping, [Collections](#), [Node Styles](#), [Production Parameters](#), [Indicator Characters](#), [Flow Mappings](#), [Block Mappings](#)

plain, [Scalars](#), [Node Styles](#), [Resolved Tags](#), [Production Parameters](#), [Indicator Characters](#), [Node Tags](#), [Empty Nodes](#), [Flow Scalar Styles](#), [Plain Style](#), [Flow Mappings](#), [Flow Nodes](#), [Block Collection Styles](#), [Block Sequences](#), [Block Mappings](#), [Block Nodes](#), [Document Markers](#), [Tag Resolution](#), [Tag Resolution](#)

scalar, [Scalars](#), [Node Styles](#), [Line Prefixes](#), [Line Folding](#), [Flow Scalar Styles](#)

sequence, [Collections](#), [Node Styles](#), [Indicator Characters](#), [Flow Sequences](#), [Flow Mappings](#)

single-quoted, [Node Styles](#), [Production Parameters](#), [Indicator Characters](#), [Flow Scalar Styles](#), [Single-Quoted Style](#)

scalar, [Node Styles](#), [Escaped Characters](#), [Empty Lines](#), [Flow Scalar Styles](#), [Literal Style](#)

single key:value pair mapping, [Keys Order](#), [Node Styles](#), [Flow Sequences](#), [Flow Mappings](#), [Block Mappings](#)

T

tab, [Prior Art](#), [Character Set](#), [White Space Characters](#), [Indentation Spaces](#), [Separation Spaces](#), [Line Prefixes](#), [Block Indentation Indicator](#)

tag, [Prior Art](#), [Tags](#), [Dump](#), [Representation Graph](#), [Nodes](#), [Tags](#), [Node Comparison](#), [Scalar Formats](#), [Loading Failure Points](#), [Resolved Tags](#), [Recognized and Valid Tags](#), [Available Tags](#), [Syntax Conventions](#), [Production Parameters](#), [Indicator Characters](#), [Miscellaneous Characters](#), [“TAG” Directives](#), [Tag Prefixes](#), [Node Properties](#), [Node Tags](#), [Flow Styles](#), [Recommended Schemas](#), [Tags](#), [Tags](#), [Other Schemas](#)

available, [Available Tags](#)

global, [Prior Art](#), [Tags](#), [Dump](#), [Tags](#), [Resolved Tags](#), [Tag Handles](#), [Tag Prefixes](#), [Node Tags](#), [Other Schemas](#)

handle, [Tags](#), [Processes](#), [Dump](#), [Indicator Characters](#), [“TAG” Directives](#), [Tag Handles](#), [Tag Prefixes](#), [Node Tags](#)

named, [Miscellaneous Characters](#), [Tag Handles](#), [Node Tags](#)
primary, [Tag Handles](#)
secondary, [Tag Handles](#)

local, [Prior Art](#), [Tags](#), [Dump](#), [Tags](#), [Resolved Tags](#), [Indicator Characters](#),
[Tag Handles](#), [Tag Prefixes](#), [Node Tags](#), [Other Schemas](#)
non-specific, [Tags](#), [Dump](#), [Loading Failure Points](#), [Resolved Tags](#), [Indicator Characters](#), [Node Tags](#), [Recommended Schemas](#), [Tag Resolution](#), [Tag Resolution](#), [Other Schemas](#)
prefix, [“TAG” Directives](#), [Tag Prefixes](#), [Node Tags](#)
property, [Resolved Tags](#), [Indicator Characters](#), [Node Tags](#)
recognized, [Recognized and Valid Tags](#)
repository, [Tags](#), [Tag Handles](#), [Other Schemas](#)

bool, [Boolean](#)
float, [Tags](#), [Floating Point](#)
int, [Tags](#), [Integer](#)
map, [Tags](#), [Generic Mapping](#)
null, [Tags](#), [Empty Nodes](#), [Null](#)
seq, [Tags](#), [Generic Sequence](#)
str, [Tags](#), [Generic String](#)

resolution, [Tags](#), [Loading Failure Points](#), [Resolved Tags](#), [Node Tags](#), [Flow Scalar Styles](#), [Recommended Schemas](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#), [Other Schemas](#)

convention, [Resolved Tags](#), [Node Tags](#), [Tag Resolution](#), [Tag Resolution](#), [Tag Resolution](#)

shorthand, [Tags](#), [Miscellaneous Characters](#), [“TAG” Directives](#), [Tag Handles](#), [Tag Prefixes](#), [Node Tags](#)
specific, [Resolved Tags](#), [Node Tags](#)
unavailable, [Load](#), [Loading Failure Points](#), [Available Tags](#)
unrecognized, [Loading Failure Points](#), [Recognized and Valid Tags](#)
unresolved, [Loading Failure Points](#), [Resolved Tags](#)
verbatim, [Node Tags](#)

trimming, [Line Folding](#)

V

value, [Dump](#), [Nodes](#), [Node Comparison](#), [Resolved Tags](#), [Indicator Characters](#), [Flow Mappings](#), [Block Mappings](#), [Generic Mapping](#), [Null](#)

Y

YAML 1.1 processing, [Line Break Characters](#), [“YAML” Directives](#)