YAML Ain't Markup Language (YAML) 1.0

Working Draft 200?-???-??

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

YAML Ain't Markup Language (YAML™) 1.0 : Working Draft 200?-???-??

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

Status of this Document

This is an intermediate working draft and is being actively revised. Hopefully the next draft will be a release candidate. TODO: Verify indentation computations (there's a +/-1 problem in block indentation). review production naming and folding descisions. Allow draft to be printed (direct DocBook -> PDF?).

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 structures of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing. Together with the Unicode [http://www.unicode.org/] standard for characters, this specification provides all the information necessary to understand YAML Version 1.0 and to construct programs that process YAML information.

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

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

Open, interoperable and readily understandable tools have advanced computing immensely. YAML was designed from the start to be useful and friendly to the people working with data. It uses printable Unicode characters, some of which provide structural information and the rest representing 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 is used for structure, colons separate pairs, and dashes are used for 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 encoding any data structure. While most programming languages can use YAML for data serialization, YAML excels in those languages that are fundamentally built around the three basic primitives. These include the new wave of agile languages such as Perl, Python, PHP, Ruby and Javascript.

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

Goals

The design goals for YAML are:

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

Prior Art

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

YAML integrates and builds upon concepts described by C [http://cm.bell-labs.com/cm/cs/cbook/index.html], Java [http://java.sun.com/], Perl [http://www.perl.org/], Python [http://www.python.org/], Ruby [http://www.ruby-lang.org/], RFC0822 [http://www.ietf.org/rfc/rfc0822.txt] (MAIL), RFC1866 [http://www.ics.uci.edu/pub/ietf/html/rfc1866.txt] (HTML), RFC2045 [http://www.ietf.org/rfc/rfc2045.txt] (MIME), RFC2396 [http://www.ietf.org/rfc/rfc2396.txt] (URI), XML [http://www.w3.org/TR/REC-xml.html], SAX [http://www.saxproject.org/] and SOAP [http://www.w3.org/TR/SOAP].

The syntax of YAML was motivated by Internet Mail (RFC0822) and remains partially compatible with that standard. Further, YAML borrows the idea of having multiple documents 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 block scoping is similar to Python's (without the ambiguities caused by tabs). Indented blocks facilitate easy inspection of a document's structure. YAML's literal scalar leverages this by enabling formatted text to be cleanly mixed within an indented structure without troublesome escaping.

YAML's double quoted scalar uses familiar C-style escape sequences. This enables ASCII representation of non-printable or 8-bit (ISO 8859-1) characters such as “**`\x3B`**”. 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 folded scalar employs an intuitive method of handling white space. In YAML, single line breaks may be folded into a single space, while empty lines represent line break characters. This technique allows for paragraphs to be word-wrapped without affecting the canonical form of the content.

YAML's core type system is based on the requirements of Perl, Python and Ruby. YAML directly supports both collection (hash, array) values and scalar (string) values. Support for common types enables programmers to use their language's native data constructs for YAML manipulation, instead of requiring a special document object model (DOM).

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

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

Relation to XML

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

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

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

Terminology

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

may	This word, or the adjective “*optional*”, mean that conforming YAML processors are permitted, but need not behave as described.
should	This word, 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	This word, or the term “*required*” or “*shall*”, mean that the behavior described is an absolute requirement of the specification.

Chapter 2. Preview

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

Collections

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

Example 2.1. Sequence of scalars (ball players)

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

Example 2.2. Mapping of scalars to scalars (player statistics)

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

Example 2.3. Mapping of 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 in-line flow styles for compact notation. The flow sequence is written as a comma separated list within square brackets. In a similar manner, the flow mapping uses curly braces. In YAML, the space after the “`-`” and “`:`” and “`:`” is mandatory.

Example 2.5. Sequence of sequences

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

Example 2.6. Mapping of mappings

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

Structures

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

Repeated nodes are first marked with the ampersand ("**&**") and then referenced with an asterisk ("*****") thereafter.

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
...
```

Example 2.9. Single document with two throwaway 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
```

The question mark indicates a complex key. Within a block sequence, mapping pairs can start immediately following the dash.

Example 2.11. Mapping between sequences

```
? # PLAY SCHEDULE
  - Detroit Tigers
  - Chicago Cubs
:
  - 2001-07-23

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

Example 2.12. Sequence key shortcut

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

Scalars

Scalar values can be written in block form using a literal style ("**|**") where all new lines count. Or they can be written with the folded style ("**>**") for content that can be word wrapped. In the folded style, newlines are treated as a space unless they are part of a blank or indented line.

Example 2.13. In literals, newlines are preserved

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

Example 2.14. In the plain scalar, newlines are treated as a space

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

Example 2.15. Folded newlines preserved for indented and blank lines

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

   63 Home Runs
   0.288 Batting Average

 What a year!
```

Example 2.16. Indentation determines scope

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

YAML's flow scalars include the plain style (most examples thus far) and quoted styles. The double quoted style provides escape sequences. Single quoted style is useful when escaping is not needed. All flow scalars can span multiple lines; intermediate whitespace is trimmed to a single space.

Example 2.17. Quoted scalars

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

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

Example 2.18. Multi-line flow scalars

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

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

Tags

In YAML, plain (unquoted) scalars are given an implicit type depending on the application. The examples in this specification use types from YAML's tag repository, which includes types like integers, floating point values, timestamps, null, boolean, and string values.

Example 2.19. Integers

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

Example 2.20. Floating point

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

Example 2.21. Miscellaneous

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

Example 2.22. Timestamps

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

Explicit typing is denoted with a tag using the exclamantion point (“**!**”) symbol. Application tags should include a domain name and may use the caret (“**^**”) to abbreviate subsequent tags.

Example 2.23. Various explicit tags

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

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

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

Example 2.24. Application specific tag

```
# Establish a tag prefix
--- !clarkevans.com,2002/graph/^shape
  # Use the prefix: shorthand for
  # !clarkevans.com,2002/graph/circle
- !^circle
  center: &ORIGIN {x: 73, y: 129}
  radius: 7
- !^line
  start: *ORIGIN
| finish: { x: 89, y: 102 }
- !^label
  start: *ORIGIN
  color: 0xFFEEBB
  text: Pretty vector drawing.
```

Example 2.25. Unordered set

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

Example 2.26. Ordered mappings

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

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

```
--- !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

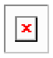
```
---
Time: 2001-11-23 15:01:42 -05:00
User: ed
Warning:
  This is an error message
  for the log file
---
Time: 2001-11-23 15:02:31 -05:00
User: ed
Warning:
  A slightly different error
  message.
---
Date: 2001-11-23 15:03:17 -05:00
User: ed
Fatal:
  Unknown variable "bar"
Stack:
  - file: TopClass.py
    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 representing native language data structures in this format. This specification defines two concepts: a class of data objects called YAML representations, and a syntax for encoding 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 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 language 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 stream of characters, in a manner friendly to humans.

Figure 3.1. YAML Overview



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

Processes

This section details the processes shown in the diagram above.

Represent

YAML representations model the data constructs from agile programming languages, such as Perl, Python, or Ruby. YAML representations view native language data objects in a generic manner, allowing data to be portable between various programming languages and implementations. Strings, arrays, hashes, and other user-defined types are supported. This specification formalizes what it means to be a YAML representation and suggests how native language objects can be viewed as a YAML representation.

YAML representations are constructed with three primitives: the sequence, the mapping and the scalar. By sequence we mean an ordered collection, by mapping we mean an unordered association of unique keys to values, and by scalar we mean any object with opaque structure yet expressible as a series of Unicode characters. When used generatively, these primitives construct 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.

YAML represents any native language data object as one of these three primitives, together with a type specifier called a *tag*. Type specifiers are either global, using a syntax based on the domain name and registration date, or private in scope. For example, an integer is represented in YAML with a scalar plus a globally scoped **`tag:yaml.org,2002/int`** tag. Similarly, an invoice object, particular to a given organization, could be represented as a mapping together with a **`tag:private.yaml.org,2002:invoice`** tag. This simple model, based on the sequence and mapping and scalar together with a type specifier, can represent any data structure independent of programming language.

Serialize

For sequential access mediums, such as an event callback API, a YAML representation must be serialized to an ordered tree. Serialization is necessary since nodes in a YAML representation may be referenced more than once (more than one incoming arrow) and since mapping keys are unordered. Serialization is accomplished by imposing an ordering on mapping keys and by replacing the second and subsequent references to a given node with place holders called aliases. The result of this process, the YAML serialization tree, can then be traversed to produce a series of event calls for one-pass processing of YAML data.

Present

YAML *character streams* (or documents) encode YAML representations into a series of characters. Some of the characters in a YAML stream represent the content of the source information, while other characters are used for presentation style. Not only must YAML character streams store YAML representations, they must do so in a manner which is human friendly.

To address human presentation, the YAML syntax has a rich set of stylistic options which go far beyond the needs of data serialization. YAML has two approaches for expressing a node's nesting, one that uses indentation to designate depth in the serialization tree and another which uses begin and end delimiters. Depending upon escaping and how line breaks should be treated, YAML scalars may be written with many different styles. YAML syntax also has a comment mechanism for annotations orthogonal to the “content” of a YAML representation. These presentation level details provide sufficient variety of expression.

In a similar manner, for human readable text, it is frequently desirable to omit data typing information which is often obvious to the human reader and not needed. This is especially true if the information is created by hand, expecting humans to bother with data typing detail is optimistic. Implicit type information may be restored using a data schema or similar mechanisms.

Parse

Parsing is the inverse process of presentation, it takes a stream of characters and produces a series of events.

Compose

Composing takes a series of events and produces a node graph representation. See completeness for more detail on the constraints composition must follow. When composing, one must deal with broken aliases and anchors, and other things of this sort.

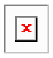
Construct

Construction converts construct YAML representations into native language objects.

Information Models

This section has the formal details of the results of the processes.

Figure 3.2. YAML Information Models



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. While imposing a order on mapping keys is necessary for flattening YAML representations to a sequential access medium, the specific ordering of a mapping should not be used to convey application level information. In a similar manner, while indentation technique or the specific scalar style is needed for character level human presentation, this syntax detail is not part of a YAML serialization nor a YAML representation. By carefully separating properties needed for serialization and presentation, YAML representations of native language information will be consistent and portable between various programming environments.

Node Graph Representation

In YAML's view, native data is represented as a directed graph of tagged nodes. Nodes that are defined in terms of other nodes are collections and nodes that are defined independent of any other nodes are scalars. YAML supports two kinds of collection nodes, sequence and mappings. Mapping nodes are somewhat tricky because its keys are considered to be unordered and unique.

Figure 3.3. YAML Representation

Nodes

A YAML representation is a rooted, connected, directed graph. By “directed graph” we mean a set of nodes and arrows, where arrows connect one node to another (a formal definition [http://www.nist.gov/dads/HTML/directedgraph.html]). Note that the YAML graph may include cycles, and a node may have more than one incoming arrow.

YAML nodes have a tag and can be of one of three kinds: scalar, sequence, or mapping. The node's tag serves to restrict the set of possible values which the node can have.

scalar	A scalar is a series of zero or more Unicode characters. YAML places no restriction on the length or content of the series.
sequence	A sequence is a series of zero or more nodes. In particular, a sequence may contain the same node more than once or it could even contain itself (directly or indirectly).
mapping	A mapping is an unordered set of key/value node pairs, with the restriction that each of the keys is unique. This restriction has non-trivial implications detailed below. YAML places no further restrictions on the nodes. In particular, keys may be arbitrary nodes, the same node may be used as a value in several 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 a *collection*. 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 for both constructing practical YAML tools and APIs and for theoretical analysis.

YAML allows several representations to be encoded to the same character stream. Representations appearing in the same character stream are independent. That is, a given node may not appear in more than one representation graph.

Tags

YAML represents type information of native objects with a simple identifier, called a *tag*. These identifiers are URIs [http://www.ietf.org/rfc/rfc2396.txt], using a subset of the "**`tag`**" URI scheme. YAML tags use only the domain based form, **`tag:domain,date:identifier`**, for example, **`tag:yaml.org,2002:str`**. YAML presentations provide several mechanisms to make this less verbose. Tags may be minted by those who own the domain at the specified date. The day must be omitted if it is the 1st of the month, and the month and day must be omitted for January 1st. The year is never omitted. Thus, each YAML tag has a single globally unique representation. More information on this URI scheme can be found at http://www.taguri.org (mirror [http://yaml.org/spec/taguri.txt]).

YAML tags can be either globally unique, or private to a single representation graph. Private tags start with **`tag:private.yaml.org,2002:`**. Clearly private tags are not globally unique, since the domain name and the date are fixed.

YAML does not mandate any special relationship between different tags that begin with the same substring. Tags ending 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 "versions" 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, **`tag:perl.yaml.org,2002:`** tags use "**`::`**" to express namespace hierarchies, **`tag:java.yaml.org,2002:`** tags use "**`.`**", etc.

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

Equality

Since YAML mappings require key uniqueness, representations must include a mechanism for testing the equality of nodes. This is non-trivial since YAML presentations allow various ways to write a given scalar. For example, the integer ten can be written as **`10`** or **`0xA`** (hex). If both forms are used as a key in the same mapping, only a YAML processor which "knows" about integer tags and their presentation 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 have a mechanism to produce a canonical form of its scalars. By canonical form, we mean a Unicode character string which represents the scalar's 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 value 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 scalar nodes are equal only when their canonical values are character-by-character equivalent. Equality of collections is defined recursively. Two sequences are equal only when they have the same 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 equal sets of keys, and each key in this set is associated with equal values in both mappings.
identity	Node equality should not be confused with node *identity*. Two nodes are identical only when they represent the same native object. Typically, this corresponds to a single memory address. During serialization [], equal scalar nodes may be treated as if they

were identical. In contrast, the separate identity of two distinct, but equal, collection nodes must be preserved.

Event Tree Serialization

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

Figure 3.4. YAML Serialization

Key Order

In the representation model, keys in a mapping do not have order. To serialize a mapping, it is necessary to impose an ordering on its keys. This order should not be used when composing a representation graph from serialized events.

In every case where node order is significant, a sequence must be used. For example, an ordered mapping can be represented by a sequence of mappings, where each mapping is a single key/value pair. YAML presentations provide convenient shorthand syntax for this case.

Anchors and Aliases

In the representation model, a node may appear in more than one collection. When serializing such nodes, the first occurrence of the node is serialized with an *anchor* and subsequent occurrences are serialized as an *alias* which specifies the same anchor. Anchors need not be unique within a serialization. When composing a representation graph from serialized events, alias nodes refer to the most recent node in the serialization having the specified anchor.

An anchored node need not have an alias referring to it. It is therefore possible to provide an anchor for all nodes in serialization. After composing a representation graph, the anchors are discarded. Hence, anchors must not be used for encoding application data.

Character Stream Presentation

YAML presentations make use of styles, comments, directives and other syntactical details. Although the processor may provide this information, these features should not be used when constructing native structures.

Figure 3.5. YAML Presentation

Node Styles

In the syntax, each node has an additional *style* property, depending on its node. There are two types of styles, *block* and *flow*. Block styles use indentation to denote nesting and scope within the presentation. In contrast, flow styles rely on explicit markers to denote nesting and scope.

YAML provides several shorthand forms for collection styles, allowing for compact nesting of collections in common cases. For compact set notation, null mapping values may be omitted. For compact ordered mapping notation, a mapping with a single key: value pair may be specified directly inside a flow sequence collection. Also, simple block collections may begin in-line rather than the next line.

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

Throwaway Comments

The syntax allows optional *comment* blocks to be interleaved with the node blocks. Comment blocks may appear before or after any node block. A comment block can't appear inside a scalar node value.

Document Directives

Each document may be associated with a set of directives. A *directive* is a key: value pair where both the key and the value are simple strings. Directives are instructions to the YAML processor, allowing for extending YAML in the future. This version of YAML defines a single directive, “**`YAML`**”. Additional directives may be added in future versions of YAML. A processor should ignore unknown directives with an appropriate warning. There is no provision for specifying private directives. This is intentional.

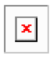
The “**`YAML`**” directive specifies the version of YAML the document adheres to. This specification defines version **`1.0`**. A version 1.0 processor should accept documents with an explicit “**`%YAML:1.0`**” directive, as well as documents lacking a “**`YAML`**” directive. Documents with a directive specifying a higher minor version (e.g. “**`%YAML:1.1`**”) should be processed with an appropriate warning. Documents with a directive specifying a higher major version (e.g. “**`%YAML:2.0`**”) should be rejected with an appropriate error message.

Completeness

The process of converting YAML information from a character stream presentation to a native data structure has several potential failure points. The character stream may be ill-formed, implicit 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 conversion.

A *partial representation* need not specify the tag of each node, and the canonical form of scalar values need not be available. This weaker representation is useful for cases of incomplete knowledge of tags used in the document.

Figure 3.6. YAML Completeness



Well-Formed

A *well-formed* character stream must match the productions specified in the next chapter. A YAML processor should reject *ill-formed* input. A processor may recover from syntax errors, but it must provide a mechanism for reporting such errors.

Resolved

It is not required that all tags in a complete YAML representation be explicitly specified in the character stream presentation. In this case, these *implicit tags* must be *resolved*.

When resolving tags, a YAML processor must only rely upon representation details, with one notable exception. It may consider whether a scalar was written in the plain style when resolving the scalar's tag. Other than this exception, the processor must not rely upon presentation or serialization details. In particular, it must not consider key order, anchors, styles, spacing, indentation or comments.

The plain scalar style exception allows unquoted values to signify numbers, dates, or other typed data, while quoted values are treated as generic strings. With this exception, a processor may match plain scalars against a set of regular expressions, to provide automatic resolution of such types without an explicit tag.

If a document contains *unresolved* nodes, the processor is unable to compose a complete representation graph. However, the processor may compose an partial representation, based on each node's kind (mapping, sequence, scalar) and allowing for unresolved tags.

Recognized and Valid

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

Available

In a given processing environment, there may 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 YAML representation may still be composed, and an application may wish to use this representation directly.

Chapter 4. Syntax

Following are the BNF productions defining the syntax of YAML character streams. The productions introduce the relevant character classes, describe the processing of white space, and then follow with the decomposition of the stream into logical chunks. To make this chapter easier to follow, production names use Hungarian-style notation:

`c-`	a production matching one or more characters starting and ending with a special character
`b-`	a production matching a single line break
`nb-`	a production matching one or more characters starting and ending with a non-break character
`s-`	a production matching one or more characters starting and ending with a space character
`ns-`	a production matching one or more characters starting and ending with a non-space character
`X-Y-`	a production matching a sequence of one or more characters, starting with an `X-` character and ending with a `Y-` character
`l-`	a production matching one or more lines (shorthand for **`i-b-`**)

Productions are generally introduced in a "bottom-up" order; more basic productions are specified before the more complex productions using them. Examples accompanying the productions list sample YAML text side-by-side with equivalent YAML text using only flow collections and double-quoted scalars. All implicit tags are resolved assuming the use of the **`!map`**, **`!seq`**, **`!str`** and **`!int`** tags.

Characters

Character Set

YAML streams use a subset of the Unicode character set. On input, a YAML processor must accept all printable ASCII characters, the space, tab, line break, and all Unicode characters beyond 0x9F. On output, a YAML processor must only produce those acceptable characters, and should also escape all non-printable Unicode characters.

The allowed character range explicitly excludes the surrogate block **`[#xD800-#xDFFF]`**, DEL **`0x7F`**, the C0 control block **`[#x0-#x1F]`**, the C1 control block **`[#x80-#x9F]`**, **`#xFFFE`** and **`#xFFFF`**. Note that in UTF-16, characters above **`#xFFFF`** are represented with a surrogate pair. When present, DEL and characters in the C0 and C1 control block must be represented in a YAML stream using escape sequences.

```
[1]   c-printable ::=   #x9 | #xA | #xD
                      | [#x20-#x7E] | #x85
                      | [#xA0-#xD7FF]
                      | [#xE000-#xFFFD]
                      | [#x10000-#x10FFFF]
```

Character Encoding

A YAML processor must support the UTF-16 and UTF-8 character encodings. If an input stream does not begin with a byte order mark, the encoding shall be UTF-8. Otherwise it shall be UTF-16 (LE or BE), as signaled by the byte order mark. Since YAML files may only contain printable characters, this does not raise any ambiguities. For more information about the byte order mark and the Unicode character encoding schemes see the Unicode FAQ [http://www.unicode.org/unicode/faq/utf_bom.html].

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

In the examples, byte order mark characters are represented as “⇔”. Note that in the interest of simplicity, a byte order mark should only appear for UTF16 encoding and that UTF32 is explicitly not supported.

Example 4.1. Byte Order Mark

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

```
# The stream contains
# no document, only
# a prefix.
```

Indicator Characters

Indicators are characters that have special semantics used to describe the structure and content of a YAML document. Indicators can be separated into two groups: Top indicators denote the structure of the character stream, while sub indicators only have special semantics in the context of a particular structure.

```
[3]      c-sequence-start ::= “[”
[4]        c-sequence-end ::= “]”
[5]       c-mapping-start ::= “{”
[6]         c-mapping-end ::= “}”
[7]      c-sequence-entry ::= “-”
[8]         c-mapping-key ::= “?”
[9]       c-mapping-value ::= “:”
[10]      c-collect-entry ::= “,”
[11]          c-throwaway ::= “#”
[12]             c-anchor ::= “&”
[13]              c-alias ::= “*”
[14]                c-tag ::= “!”
[15]            c-literal ::= “|”
[16]             c-folded ::= “>”
[17]       c-single-quote ::= “'”
[18]       c-double-quote ::= “"”
[19]          c-directive ::= “%”
[20]           c-reserved ::= “@” | “`”
[21]     c-top-indicators ::=   “[” | “]” | “{” | “}”
                              | “-” | “?” | “:” | “,”
                              | “#” | “&” | “*” | “!”
                              | “|” | “>” | “'” | “"”
                              | “%” | “@” | “`”
```

Example 4.2. Top Indicator Characters

```
--- %YAML:1.0
# top level block map:
flow seq: &anchor [ one, 1,200 ]
flow map: { one: 1, 1,200: 2 }
block seq: !seq
- one
- -1,200
? alias
: *anchor
literal: |
  #!/usr/bin/perl
  print 'Hello, world!', "\n";
folded: >
  Line-wrapped
  paragraphs &
  WIKI-style
  formatting:

    * false ⇒ anything,
    * 1,200 > -7

single quoted:
  "*p[1] is a C expression"
double quoted:
  '"s" is a C string'
...
# Legend:
#   c-top-indicators
#   not top indicators
```

```
--- %YAML:1.0
!map {
  !str "flow seq": &anchor
    !seq [ !str "one", !int "1200" ],
  !str "flow map":
    !map {
      !str "one": !int "1",
      !int "1200": !int "2",
    }
  !str "block seq":
    !seq [ !str "one", !int 1200 ]
  !str "alias": *anchor
  !str "literal": !str
    "#!/usr/bin/perl\n\
    print 'Hello, world!', \"\\n\";\n"
  !str "folded": !str
    "Line-wrapped paragraphs &\
    WIKI-style formatting:\n\
    \n\
    \  * false \u21d2 anything,\n\
    \  * 1,200 > -7\n"
  !str "single quoted":
    !str "*p[1] is a C expression"
  !str "double quoted":
    !str "\"s\" is a C string"
}
...
```

```
[22]            c-domain ::= “.”
[23]              c-date ::= “,”
[24]              c-path ::= “/”
[25]            c-prefix ::= “^”
[26]            c-escape ::= “\”
[27]             c-strip ::= “-”
[28]              c-keep ::= “+”
[29]    c-sub-indicators ::=   “.” | “,” | “/” | “^”
                             | “\” | “-” | “+”
```

Example 4.3. Sub Indicator Characters

```
---
!clarkevans.com,2002/graph/^shape
- !^label
  text: "Multi\nLine"

- !^label
  text: |+
    To /tmp or to c:\temp,
    that is the question...

- !^label
  text: >-
    -1.0^2 = +2,000/2,000

...
# Legend:
#   c-sub-indicators
#   not sub indicators
```

```
---
!clarkevans.com,2002/graph/shape [
  !clarkevans.com,2002/graph/label {
    !str "text":
      !str "Multi\nLine"
  },
  !clarkevans.com,2002/graph/label {
    !str "text":
      !str "To /tmp or to c:\\temp,\n\
          that is the question...\n\n"
  },
  !clarkevans.com,2002/graph/label {
    !str "text":
      !str "-1.0^2 = +2,000/2,000"
  }
]
...
```

Line Break Characters

The Unicode standard defines several line break characters. These line breaks can be grouped into two categories. Specific line breaks have well-defined semantics for breaking text into lines and paragraphs, and must be preserved by the YAML processor. Generic line breaks do not carry a meaning beyond “ending a line” and are *normalized* by the YAML processor: inside scalar text content, each such line break must be translated into a single LF character (this does not apply to escaped characters). This normalization functionality is indicated by the use of the **`b-as-line-feed`** production defined below. Outside scalar text content, YAML allows any line break to be used to terminate lines, and in most cases also allows such line breaks to be preceded by trailing comment characters. On output, a YAML processor is free to present line breaks using whatever convention is most appropriate, though specific line breaks must be preserved in scalar content. These rules are compatible with Unicode's newline guidelines [http://www.unicode.org/unicode/reports/tr13/].

```
[30]                b-line-feed ::= #xA /* LF */
[31]          b-carriage-return ::= #xD /* CR */
[32]                     b-next ::= #x85 /* NEL */
[33]           b-line-separator ::= #x2028 /* LS */
[34]      b-paragraph-separator ::= #x2029 /* PS */
[35]                     b-char ::=   b-line-feed
                                    | b-carriage-return
                                    | b-next
                                    | b-line-separator
                                    | b-paragraph-separator
[36]                  b-generic ::=   ( b-carriage-return b-line-feed )
                                    | b-carriage-return
                                    | b-line-feed
                                    | b-next
```

```
[37]                  b-specific ::=   b-line-separator
                                     | b-paragraph-separator
[38]                       b-any ::= b-generic | b-specific
[39]              b-as-line-feed ::= b-generic
[40]                b-normalized ::= b-as-line-feed | b-specific
```

Example 4.4. Line Break Characters

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

```
---
Generic line break (no glyph)
Generic line break (glyphed)↓
Line seperator⇓
Paragraph seperator¶
...
# Legend:
#   b-generic
#   b-specific
```

```
--- !str
"Generic line break (no glyph)\n\
Generic line break (explicit)\n\
Line seperator\u2028\
Paragraph seperator\u2029"
...
```

Miscellaneous Characters

YAML makes use of the following common character range definitions:

```
[41]                     nb-char ::= c-printable - b-char
[42]                      s-char ::=   #x9 /* TAB */
                                     | #x20 /* SP */
[43]                     ns-char ::= nb-char - s-char
[44]             ns-ascii-letter ::=   [#x41-#x5A] /* A-Z */
                                     | [#x61-#x7A] /* a-z */
[45]            ns-decimal-digit ::= [#x30-#x39] /* 0-9 */
[46]                ns-hex-digit ::=   ns-decimal-digit
                                     | [#x41-#x46] /* A-F */
                                     | [#x61-#x66] /* a-f */
[47]                ns-word-char ::=   ns-decimal-digit
                                     | ns-ascii-letter | “-”
```

Escaped Characters

Non-printable characters must be encoded in the YAML character stream using escape sequences. YAML escape sequences are based on the “\” notation used by most modern computer languages. To allow long lines to be broken in arbitrary locations, an escaped line break is completely ignored. Escape sequences are only interpreted in two contexts, in double quoted scalars and in tags. In all other contexts, the “\” character has no special meaning.

```
[48]                 c-b-escaped ::= “\” b-any /* ignored */
[49]               ns-esc-escape ::= “\” “\”
[50]         ns-esc-double-quote ::= “\” “"”
[51]                  ns-esc-bel ::= “\” “a”
[52]            ns-esc-backspace ::= “\” “b”
```

```
[53]                  ns-esc-esc ::= “\” “e”
[54]            ns-esc-form-feed ::= “\” “f”
[55]            ns-esc-line-feed ::= “\” “n”
[56]               ns-esc-return ::= “\” “r”
[57]                  ns-esc-tab ::= “\” “t”
[58]         ns-esc-vertical-tab ::= “\” “v”
[59]                ns-esc-caret ::= “\” “^”
[60]                 ns-esc-null ::= “\” “0”
[61]                ns-esc-space ::= “\” #20 /* SP */
[62]    ns-esc-non-breaking-space ::= “\” “_”
[63]                 ns-esc-next ::= “\” “N”
[64]       ns-esc-line-separator ::= “\” “L”
[65]  ns-esc-paragraph-separator ::= “\” “P”
[66]                ns-esc-8-bit ::= “\” “x” ( ns-hex-digit x 2 )
[67]               ns-esc-16-bit ::= “\” “u” ( ns-hex-digit x 4 )
[68]               ns-esc-32-bit ::= “\” “U” ( ns-hex-digit x 8 )
[69]             ns-esc-sequence ::=   ns-esc-escape
                                     | ns-esc-double-quote
                                     | ns-esc-bel
                                     | ns-esc-backspace
                                     | ns-esc-esc
                                     | ns-esc-form-feed
                                     | ns-esc-line-feed
                                     | ns-esc-return
                                     | ns-esc-tab
                                     | ns-esc-vertical-tab
                                     | ns-esc-caret
                                     | ns-esc-null
                                     | ns-esc-space
                                     | ns-esc-non-breaking-space
                                     | ns-esc-next
                                     | ns-esc-line-separator
                                     | ns-esc-paragraph-separator
                                     | ns-esc-8-bit
                                     | ns-esc-16-bit
                                     | ns-esc-32-bit
```

Example 4.5. Escaped Characters

```
--- !foo.bar/\^baz
"Fun with \\
\" \a \b \e \f \↓
\n \r \t \v \0 \⇓
\  \_ \N \L \P \¶
\x41 \u0041 \U00000041"
...
# Legend:
#   ns-esc-sequence
```

```
--- !foo.bar/\x5ebaz
"Fun with \x5C
\x22 \x07 \x08 \x1B \0C
\x0A \x0D \x09 \x0B \x00
\x20 \xA0 \x85 \u2028 \u2029
A A A"
...
```

Example 4.6. Invalid Escaped Characters

```
---
"Bad escapes: \c \xq-"
...
# Legend:
#   invalid escape sequences
```

```
ERROR:
- c is an invalid escaped character.
- q and - are invalid hecadecimal
  digits (applies to \x, \u and \U).
```

Syntax Primitives

Production Context

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

Indentation: `n` or `m`	Since the YAML stream depends upon indentation level to delineate blocks, many productions are parameterized by it. In some cases, the notations **`production(<n)`**, **`production(≤n)`** and **`production(>n)`** are used; these are shorthands for "**`production(m)`** for some specific `m`" where $0 \le m < n$, $0 \le m \le n$ and $m > n$, respectively. Similarly, the **`production(≥0)`** notation is used to indicate an arbitrary indentation level, and is a shorthand for "**`production(m)`** for some specific `m`" where $m \ge 0$.
Context: `c`	YAML supports two main contexts, *block* and *flow*. In the block contexts, indentation is used to delineate structure. Due to the fact that the "**`-`**" sequence entry indicator is perceived as an indentation character, some productions distinguish between the *block-in* context (inside block sequences) and the *block-out* context (outside block sequences). In the flow contexts, explicit markers are used to delineate structure. As plain scalars have no such markers, they are the most context sensitive constructs, distinguishing between being nested inside a flow collection (*flow-in* context) or being outside one (*flow-out* context). YAML also provides for a terse and intuitive syntax using plain scalars as flow mapping keys (*flow-key* context). Such scalars are the most restricted, for readability and implementation reasons.
Style: `s`	Typically, each presentation style is specified using a distinct set of productions. However, for some scalar styles it is better have a single productions set parameterized by the style rather than having two mostly-identical sets of productions. Thus, quoted scalar productions may be *single* or *double*, and block scalar productions may be *literal* or *folded*.
Chomping (trailing line breaks): `t`	Block scalars offer three possible mechanisms for *chomping* any trailing line breaks: *strip*, *clip* and *keep*.

Indentation Levels

In a YAML character stream, structure is often determined from indentation, where indentation is defined as a line break character followed by zero or more space characters. Tab characters are not allowed in indentation since different systems treat tabs differently. To maintain portability, YAML's tab policy is conservative; they must not be used. Note that most modern editors may be configured so that pressing the tab key results in the insertion of an appropriate number of spaces.

Example 4.7. Invalid Use of Tabs

In the examples, tab characters are represented as the glyph “→”.

```
---
tabs:→# → Dos and don'ts.
→- Indented by a tab (→).
 → - Mixed indentation.
...
```

```
ERROR:
  Tabs must not appear in
  indentation. They may
  appear anywhere else - in
  content, comments, etc.
```

The indentation level is always non-zero, except for the top level node of each document. This node is commonly indented by zero spaces (not indented). When such top level block scalar node is not indented, all lines up to the next document separator, document terminator, or end of the stream are assumed to be content lines. Note this includes lines beginning with a “**#**” character. In general, a node must be more indented than its parent node. All sibling nodes must use the exact same indentation level, however the content of each sibling node could be further indented independently.

The “**-**” sequence entry, “**?**” mapping key and “**:**” mapping value indicators are perceived by people to be part of the indentation. Hence the indentation rules are slightly more flexible when dealing with these indicators. First, a block sequence need not be indented relative to its parent node, unless that node is also a block sequence. Second, compact in-line notations allow a nested collection to begin immediately following the indicator (where the indicator is counted as part of the indentation), providing for a compact and intuitive collection nesting notation.

Indentation is used exclusively to delineate structure and is otherwise ignored; in particular, indentation characters must never be considered part of the document's content.

[70] `s-indentation(n) ::= #x20 x n`

Example 4.8. Indentation Spaces

```
 # Indented text and empty comments
 
---
No indent spaces:
 By one space: >
 By four
   spaces
 Flow style:→[
   one,
 two
  → three
   ]
...
# Legend:
#   s-indentation
#   not indentation spaces
```

```
---
!map {
  !str "No indent spaces":
    !map {
      !str "By one space":
        !str "By four\n  spaces",
      !str "Flow style":
        !seq [
          !str "one",
          !str "two three"
        ]
    }
}
...
```

Throwaway comments

Throwaway comments have no effect whatsoever on the document's representation graph. The usual purpose of a comment is to communicate between the human maintainers of the file. A typical example is comments in a configuration file. An explicit throwaway comment is marked by a “**#**” indicator and always spans to the end of a line. Explicit comments can be indented on their own line, or may, in some cases, follow other syntax elements (with leading spaces). Outside text content, empty lines or lines containing only white space are taken to be implicit throwaway comment lines. A throwaway comment may appear before a document's top level node or following any node. It may not appear inside a scalar node, but may precede or follow it.

```
[71]          s-discarded-spaces ::= s-char+
[72]        c-nb-throwaway-text ::= “#” nb-char*
[73]              b-ignored-any ::= b-any
[74]      c-b-throwaway-comment ::= c-nb-throwaway-text? b-ignored-any
[75]      s-b-seperated-comment ::= ( s-discarded-spaces
                                      c-nb-throwaway-text? )?
                                    b-ignored-any
```

Example 4.9. Trailing Comments

```
---
    # Comment Line↓
Block: |     # Comment↓
  The
  # of elements
 # Is commented↓
Flow:     # Comment↓
  The  # of elements
  # Is commented↓
...
# Legend:
#   c-b-throwaway-comment
#   s-b-seperated-comment
#   c-nb-throwaway-text
#   not a comment
```

```
---
!map {
  !str "Block":
    !str "The # of elements",
  !str "Flow":
    !str "The # of elements"
}
...
```

```
[76]            l-empty-comment(n) ::= s-indentation(n)
                                       b-ignored-any
[77]             l-text-comment(n) ::= s-indentation(n)
                                       c-b-throwaway-comment
[78]                     l-comment ::=   l-empty-comment(≥0)
                                       | l-text-comment(≥0)
```

Example 4.10. Comment Lines

```
↓
# Comment⇓
---
# YAML can be easily quoted:
Quoted YAML: |+
# YAML quoted as
# literal text
↓
  key: value
↓
# Quoting need not be indented:
↓
--- |
# YAML quoted as
# literal text
↓
key: value
↓
...
# Legend:
#   l-text-comment
#   l-empty-comment
#   not a comment line
```

```
---
!map {
  !str "Quoted YAML":
    !str "# YAML quoted as\n\
          # literal text\n\n\
          key: value\n\n"
}
...
---
!str "# YAML quoted as\n\
      # literal text\n\n\
      key: value\n\n"
...
```

Seperation Spaces

When tokens may be separated by white space, YAML usually allows ending the line (with an optional trailing comment) and continuing in the next (indented) line. Simple keys, however, require separation spaces to be confined within the current line. Space separation functionality is indicated by the use of the **`s-seperate-spaces(n,c)`** production. Seperation spaces are used exclusively to delineate structure and are otherwise ignored; in particular, such characters must never be considered part of the document's content.

```
[79]       s-seperate-spaces(n,c) ::= c = block-out ⇒ s-seperate-span-spaces(n)
                                      c = block-in  ⇒ s-seperate-span-spaces(n)
                                      c = flow-out  ⇒ s-seperate-span-spaces(n)
                                      c = flow-in   ⇒ s-seperate-span-spaces(n)
                                      c = flow-key  ⇒ s-discarded-spaces
[80]    s-seperate-span-spaces(n) ::=   s-discarded-spaces
                                      | ( s-b-seperated-comment
                                          l-comment*
                                          s-indentation(n) s-discarded-spaces?
                                        )
```

Example 4.11. Seperation Spaces

```
--- %YAML:1.0 !map
block sequence: &anchor !seq
-→!str |-
 block (literal) entry
- flow (plain) entry
flow sequence :→ # Two entries
    [→entry ,⇓
    entry]
? >-
    flow mapping
:¶
    # Key:
    {key:↓
    # Value:
    value,}
...
# Legend:
#   s-seperate-spaces
```

```
--- %YAML:1.0
!map {
  !str "block sequence":
    !seq [
      !str "block (literal) entry",
      !str "flow (plain) entry",
    ],
  !str "flow sequence":
    !seq [
      !str "entry",
      !str "entry",
    ],
  !str "flow mapping":
    !map {
      !str "key":
        !str "value"
    }
}
...
```

Line Folding

Line folding allows long lines to be broken for presentation, while retaining the original semantics of a single long line. When folding is done, any line break ending an empty line is preserved, as are any specific line breaks. Hence, folding only applies to generic line breaks that end non-empty content lines. If the following line is not empty, the generic line break is converted to a single space (**`#x20`**). If the following line is empty, the generic line break is ignored. Folding functionaility is implied by using the b-as-space and b-ignored-generic productions.

```
[81]                  b-as-space ::= b-generic
[82]           b-ignored-generic ::= b-generic
[83]      b-l-folded-specific(n,c) ::= b-specific
                                       l-empty-line(n,c)*
[84]      b-l-folded-as-space(n,c) ::= b-as-space
[85]       b-l-folded-trimmed(n,c) ::= b-ignored-generic
                                       l-empty-line(n,c)+
[86]         b-l-folded-break(n,c) ::=   b-l-folded-specific(n,c)
                                       | b-l-folded-as-space(n,c)
                                       | b-l-folded-trimmed(n,c)
```

Example 4.12. Folded Line Breaks

```
---
Flow:
 First↓
 line↓
↓
 Second↓
 line¶
 ↓
 Third↓
 line
Block: >
 First↓
 line↓
↓
 Second↓
 line¶
 ↓
 Third↓
 line
...
# Legend:
#   b-l-folded-as-space
#   b-l-folded-trimmed
#   b-l-folded-specific
```

```
---
!map {
  !str "Flow":
    !str "First line\
          Second line\P\n\
          Third line",
  !str "Block":
    !str "First line\
          Second line\P\n\
          Third line",
}
...
```

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

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

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

```
[87]         i-s-empty-line(n,c) ::= c = block-out  ⇒ i-s-empty-line-block(n)
                                     c = block-in   ⇒ i-s-empty-line-block(n)
                                     c = flow-out   ⇒ i-s-empty-line-flow(n)
                                     c = flow-in    ⇒ i-s-empty-line-flow(n)
[88]     i-s-empty-line-block(n) ::= s-indentation(≤n)
[89]      i-s-empty-line-flow(n) ::=   s-indentation(≤n)
                                     | ( s-indentation(n)
                                         s-discarded-spaces )
[90]       l-empty-specific(n,c) ::= i-s-empty-line(n,c)
                                     b-specific
[91]        l-empty-generic(n,c) ::= i-s-empty-line(n,c)
                                     b-as-line-feed
[92]           l-empty-line(n,c) ::=   l-empty-specific(n,c)
                                     | l-empty-generic(n,c)
```

Example 4.13. Folded Empty Lines

```
---
Block (folded): >
  Some
↓
⇓
↓
⇓
  Text
Flow (plain):
  Some
↓
⇓
↓
⇓
  Text
...
# Legend:
#   l-empty-line
#   s-indentation
#   not empty line
```

```
---
!map {
  !str "Block (folded)":
    !str "Some\n\L\n \LText\n"
  !str "Flow (plain)":
    !str "Some\n\L\n\LText"
}
...
```

Block Line Chomping

YAML allows empty comment lines to follow a block scalar content, in order to visually separate it from the following node. To allow the representation of trailing empty lines in the content of block scalars, YAML supports three different “chomping” mechanisms for such content:

strip	The line break character of the last non-empty line, if any, and any trailing empty lines are not considered to be a part of the scalar's content. This behavior is specified using the “`-`” chomp indicator.

clip	The line break character of the last non-empty line, if any, is considered to be a part of the scalar's content. Any trailing empty lines are not considered to be a part of the value. This is the default behavior.
keep	The line break character of the last non-empty line, if any, and any trailing empty lines are considered to be a part of the scalar's content. This behavior is specified using the “**+**” chomp indicator.

Block scalars are clipped, unless an explicit chomp indicator is specified. When this functionality is implied, the b-chomped-break(t) and l-trail-chomped(n,t) productions are used.

```
[93]     b-chomped-break(t) ::= t = strip ⇒ b-chomped-strip
                                t = clip  ⇒ b-chomped-keep
                                t = keep  ⇒ b-chomped-keep
[94]        b-chomped-strip ::= b-ignored-any
[95]         b-chomped-keep ::= b-normalized
```

Example 4.14. Final Chomped Lines Breaks

```
---
# Clipped line breaks:
clip: |
  Text line↓
  ↓
 ⇓
¶
 # Stripped line breaks:
↓
strip: >-
  Text line¶
  ↓
 ⇓
¶
# Keep everything:
⇓
keep: |+
  Text line⇓
  ↓
 ⇓
¶
 # That's all.
¶
...
# Legend:
#   b-chomped-strip
#   b-chomped-keep
#   not chomped line break
```

```
---
!map {
  !str "clip":
    !str "Text line\n",
  !str "strip":
    !str "Text line",
  !str "keep":
    !str "Text line\L\n\L\P"
}
...
```

```
[96]     l-trail-comments ::= l-text-comment(<n) l-comment*
```

```
[97]        l-trail-chomped(n,t) ::= t = strip ⇒ l-trail-chomped-strip(n)
                                     t = clip  ⇒ l-trail-chomped-strip(n)
                                     t = keep  ⇒ l-trail-chomped-keep(n)
[98]    l-trail-chomped-strip(n) ::= l-empty-comment(n)*
                                     l-trail-comments?
[99]     l-trail-chomped-keep(n) ::= l-empty-line(n,block)*
                                     l-trail-comments?
```

Example 4.15. Trailing Chomped Empty Lines

```
---
# Clip trailing lines:
clip: |
  Text line↓
↓
⇓
¶
 # Strip final break:
↓
strip: >-
  Text line¶
↓
⇓
¶
# Keep everything:
⇓
keep: |+
  Text line⇓
↓
⇓
¶
 # That's all.
¶
...
# Legend:
#   l-trail-chomped-strip
#   l-trail-chomped-keep
#   not trailing empty line
```

```
---
!map {
  !str "clip":
    !str "Text line\n",
  !str "strip":
    !str "Text line",
  !str "keep":
    !str "Text line\L\n\L\P"
}
...
```

Nodes

The YAML character stream consists mainly of a sequence of nodes, each presenting a single serialization node. Each node may have two optional properties, anchor and tag, in addition to its content.

Node Anchor

The anchor node property marks a node for future reference. Anchors are presented in the character stream using the anchor indicator “**&**”. An alias node can then be used to indicate additional inclusions of the anchored node by specifying its anchor. An anchored node need not be referenced by any alias node; in particular, it is valid for all nodes to be anchored. Note that the anchor is not part of the YAML representation encoded in the character stream. Specifically, a YAML processor must not use the anchor's name when composing the representation, and the processor need not preserve the anchor once the representation is composed.

```
[100]    c-ns-anchor-property ::= “&” ns-anchor-name
[101]          ns-anchor-name ::= ns-char+
```

Example 4.16. Anchor Property

```
# Clipped line breaks:
first occurence: &id!12[3#4]
    Some text
second occurence: *id!12[3#4]
...
# Legend:
#   c-ns-anchor-property
#   ns-anchor-name
#   not anchor property
```

```
---
!map {
  !str "first occurence":
    &anchor !str "Some text\n",
  !str "second occurence":
    *anchor
}
...
```

Node Tag

The tag node property identifies the type of data presented by the node through a URI. Unlike anchors, tags are part of the document's representation graph. The YAML processor is responsible for resolving tags which are not present in the character stream. Tags are presented in the character stream using the tag indicator, “**!**”. YAML tags are written using a shorthand syntax rather than the full URI notation. In addition, YAML provides a prefix mechanism for compact tag presentation,

Example 4.17. Tag Property

```
---
explicit string: !str '12'
explicit integer: !int '12'
...
# Legend:
#   c-ns-tag-property
```

```
---
!map {
  !str "explicit string":
    !str "12",
  !str "explicit int":
    !int "12"
}
...
```

Tag Escaping

YAML allows arbitrary Unicode characters to be used in a tag by using escape sequences. The processor must expand such escape sequences before reporting the tag to the application. It is an error for a tag to contain invalid escape sequences.

Sometimes it may be helpful for a YAML tag to be expanded to its full URI form. A YAML processor may provide a mechanism to perform such expansion. Since URIs support a limited ASCII-based character set, this expansion requires all characters outside this set to be encoded in UTF-8 and the resulting bytes to be encoded using “**%**” notation with upper-case hexadecimal digits. Further details on the URI encoding requirements are given in RFC2396 [http://www.ietf.org/rfc/rfc2396.txt].

```
[102]                  ns-tag-char ::=  ns-esc-sequence
                                      | ( ns-char - “\” - “^” )
```

Example 4.18. Tag Escaping

```
---
# The following nodes have
# the same tag URI:
# 'tag:domain.tld,2002/a%3C%0A%25b'.
- !domain.tld,2002/a<\n%b a
- !domain.tld,2002/a\x3c\x0A%b b
...
# Legend:
#   Unconventional ns-tag-chars
```

```
---
!seq [
  !domain.tld,2002/a\x3c\x0A%b "a"
  !domain.tld,2002/a\x3c\x0A%b "b"
]
...
```

Tag Prefixing

YAML provides a convenient prefix mechanism for the common case where a node and (most of) its descendants have tags whose shorthand forms share a common prefix. If a node's tag property is of the form `prefix^suffix`, the “`^`” character is discarded from the tag. If a descendant node's tag property is of the form `^foo`, it is treated as if it was written `prefixfoo` where `prefix` comes from the most recent ancestor that established a prefix. Note that this mechanism is purely syntactical and does not imply any additional semantics. In particular, the prefix must not be assumed to be an identifier for anything. It is possible to include a “`^`” character in a tag by escaping it. It is an error for a node's tag property to contain more than one unescaped “`^`” character, or for the tag property to begin with “`^`” unless the node is a descendant of an ancestor that established a tag prefix.

```
[103]            c-ns-tag-property ::= “!”
                                      ( c-ns-private-tag
                                      | ns-global-tag
                                      | ( Prefix-of-above?
                                          “^”
                                          Suffix-of-above ) )
```

Example 4.19. Private Tags

```
# The tag URI is:
# 'tag:domain.tld,2002:invoice'
--- !domain.tld,2002/^invoice
  # The tag URI is:
  # 'tag:yaml.org,2002:seq'.
  customers: !seq
    # The tag URI is:
    # 'tag:domain.tld,2002:customer'
    - !^customer
      given : Chris
      family : Dumars
...
# Legend:
#   prefix
#   c-prefix
#   suffix
#   neither
```

```
---
!domain.tld,2002/invoice {
  !str "customers":
    !seq [
      !domain.tld,2002/customer {
        !str "given":
          !str "Chris",
        !str "family":
          !str "Dumars"
      }
    ]
}
...
```

Tag Shorthands

To increase readability, YAML does not use the full tag URI notation in the character stream. Instead, it provides several shorthand notations for different groups of tags. If a tag may be written using more than one shorthand, the shortest format must be used. A processor need not expand shorthand tags to a full URI form. However, in such a case the processor must still perform escaping. These rules ensure that each tag's shorthand is globally unique.

- If a tag property is of the form **`!`**`foo`, it is a shorthand for the private tag URI **`tag:private.yaml.org,2002:`**`foo`. The semantics of private tags may be different in each document.

[104] `c-ns-private-tag ::= “!” ns-tag-char+`

Example 4.20. Private Tags

```
# In both documents, the tag URI is:
# 'tag:private.yaml.org,2002:ball'
# However, the tag semantics in
# each document are unrelated.
---
pool: !!ball
  number: 8
---
bearing: !!ball
  material: steel
...
# Legend:
#   c-ns-private-tag
```

```
---
!map {
  !str "pool":
    !!ball {
      !str "number":
        !int "8"
    }
  }
...
---
!map {
  !str "bearing":
    !!ball {
      !str "material":
        !str "steel"
    }
}
...
```

- If a tag property `foo` contains no “`/`” and no “`:`” characters, it is a shorthand for the tag URI **`tag:yaml.org,2002:`**`foo`. The **`yaml.org`** domain is used to define the core and universal YAML data types.

```
[105]    ns-core-tag ::= ( ns-tag-char - “:” - “/” - “!” )
                         ( ns-tag-char - “:” - “/” )*
```

Example 4.21. Core Tags

```
# The URI is:
# 'tag:yaml.org,2002:str'
--- !seq
- !str Unicode string
...
# Legend:
#   ns-core-tag
```

```
---
!seq [
  !str "Unicode string"
]
...
```

- If the tag property is of the form `vocabulary/foo` where `vocabulary` is a single word, it is a shorthand for the tag URI **`tag:`**`vocabulary`**`.yaml.org,2002:`**`foo`. Each domain `vocabulary`**`.yaml.org`** is used for tags specific to the given vocabulary, such as a particular programming language.

```
[106]    ns-vocabulary-tag ::= ns-word-char+ “/” ns-tag-char*
```

Example 4.22. Vocabulary Tags

```
---
# The URI is:
# 'tag:perl.yaml.org,2002:Text::Tabs'
- !perl/Text::Tabs {}
...
# Legend:
#   ns-vocabulary-tag
```

```
---
!seq [
  !perl/Text::Tabs {
  }
]
...
```

Example 4.23. Invalid Vocabulary Tags

```
---
- !private/ball ""
...
```

```
ERROR:
  To keep shorthands unique, use the
  shortest applicable form (!!ball).
```

- Otherwise, the tag property must be of the form `domain,date/foo`, which is a shorthand for the tag URI **`tag:`**`domain,date/foo`. To ensure uniqueness, the day must be omitted if it is the 1st of the month, and the month and day must be omitted for January 1st. Such tags may be freely minted at any time by the owners of the domain at the specified date.

```
[107]      ns-domain-year ::= ns-decimal-digit x 4
[108]     ns-domain-month ::= ns-decimal-digit x 2
[109]       ns-domain-day ::= ns-decimal-digit x 2
[110]       ns-domain-tag ::= ns-word-char+
                              ( “.” ns-word-char+ )
                              “,” ns-domain-year
                              ( “-” ns-domain-month
                                ( “-” ns-domain-day )? )?
                              “/” ns-tag-char*
```

Example 4.24. Domain Tags

```
---
# The URI is:
# 'tag:clarkevans.com,2003-02:graph'
- !clarkevans.com,2003-02/graph
...
# Legend:
#   ns-domain-tag
```

```
---
!seq [
  !clarkevans.com,2003-02/graph ""
]
...
```

Example 4.25. Invalid Domain Tags

```
---
- !clarkevans.com,2003-02-01/graph ""
- !clarkevans.com,2003-01/graph ""
...
```

```
ERROR:
  To keep shorthands unique, it is
  forbidden to explicitly specify
  the 1st of the month or January 1st.
```

- The above are all the valid tag shorthand notations. In particular, YAML forbids using a URI scheme other than "**`tag:`**", and the complete tag URI must not be presented.

```
[111]         ns-global-tag ::=   ns-core-tag
                                | ns-vocabulary-tag
                                | ns-domain-tag
```

Example 4.26. Invalid Tag Presentation

```
---
- !http://www.yaml.org/bing invalid
- !tag:yaml.org,2002:str invalid
...
```

```
ERROR:
 Only the tag shorthand is allowed
 in the character stream. URIs,
 including tag: URIs, are forbidden.
```

Node Properties

Node properties may be specified in any order before the node's content, and any or both may be omitted from the presentation stream. A node without an anchor can not be referenced by alias nodes. A node without a tag is subject to tag resolution.

```
[112]   c-ns-properties(n,c) ::=   ( c-ns-tag-property
                                     ( s-seperate-spaces(n,c)
                                       c-ns-anchor-property )? )
                                 | ( c-ns-anchor-property
                                     ( s-seperate-spaces(n,c)
                                       c-ns-tag-property )? )
```

Example 4.27. Node Properties

```
--- !seq
anchored: &anchor
  !str Some text
!str alias: *anchor
...
# Legend:
# c-ns-properties
# not properties
```

```
---
!seq [
  !str "anchored":
    !str &anchor "Some text",
  !str "alias":
    *anchor
]
...
```

Node Content

Node content may be presented in either a flow or a block style. Block content always extends to the end of a line and uses indentation to denote structure, while flow content ends at some non-space character within a line and uses indicators to denote structure. Each kind of content (scalar, sequence or mapping) provides several presentation styles. Scalar content may be presented in one of two block styles and three flow styles. In contrast, collection content may be presented in a single flow style or a single block style. However, each of the collection kinds provides several compact syntax forms for common cases. The following figure lists all the combinations of content kind and style:

Figure 4.1. Kind/Style Combinations

```
[113]         ns-flow-scalar(n,c) ::=   c-double-quoted(n,c)
                                      | c-single-quoted(n,c)
                                      | ns-plain(n,c)
[114]      c-flow-collection(n,c) ::=   c-flow-sequence(n,c)
                                      | c-flow-mapping(n,c)
[115]        ns-flow-content(n,c) ::=   ns-flow-scalar(n,c)
                                      | c-flow-collection(n,c)
```

Example 4.28. Flow Node Content

```
---
flow styles:
  scalars:
    plain: !str some text↓
    quoted:
      single: 'some text'↓
      double: "some text"↓
  collections:
    sequence: !seq [ !str entry,
      # Mapping entry:↓
        key: value ]↓
    mapping: { key: value }↓
...
Legend:
#    ns-flow-scalar
#    c-flow-collection
#    not content
```

```
---
!map {
  !str "flow styles":
    !map {
      !str "scalars":
        !map {
          !str "plain":
            !str "some text",
          !str "quoted":
            !map {
              !str "single":
                !str "some text",
              !str "double":
                !str "some text"
            }
        },
      !str "collections":
        !map {
          !str "sequence":
            !seq [
              !str "entry",
              !map {
                !str "key":
                  !str "value"
              }
            ],
          !str "mapping":
            !map {
              !str "key":
                !str "value"
            }
        }
    },
}
...
```

```
[116]         c-l-block-scalar(n) ::=   c-l-folded(n)
                                      | c-l-literal(n)
```

```
[117]   c-l-block-collection(n,c) ::= c-b-throwaway-comment
                                      l-comment*
                                      ( l-block-sequence(n,c)
                                      | l-block-mapping(n) )
[118]      c-l-block-content(n,c) ::=   c-l-block-scalar(n)
                                      | c-l-block-collection(n,c)
```

Example 4.29. Block Node Content

```
---
block styles:
  scalars:
    literal: |
      #!/usr/bin/perl
      print "Hello, world!\n";↓
    folded: >
      Which of the following is the
      currency used in Albania?
        A [ ] Kwacha
        B [ ] Lek
      Etc.↓
  collections: !seq
    sequence: !seq # Entry:↓
      - entry
      # Mapping entry:↓
      - key: value↓
    mapping: !map↓
      key: value↓
...
Legend:
#   c-l-block-scalar
#   c-l-block-collection
#   not content
```

```
---
!map {
  !str "block styles":
    !map {
      !str "scalars":
        !map {
          !str "literal":
            !str "#!/usr/bin/perl\n\
              print \"Hello,
              world!\\n\";\n",
          !str "folded":
            !str "Which of the
              following is the
              currency used in
              Albania?\n\
              \  A [ ] Kwacha\n\
              \  B [ ] Lek\n\
              Etc.\n"
        },
      !str "collections":
        !map {
          !str "sequence":
            !seq [
              !str "entry",
              !map {
                !str "key":
                  !str "value"
              }
            ],
          !str "mapping":
            !map {
              !str "key":
                !str "value"
            }
        }
    }
}
...
```

Alias Node

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 refers to the most recent preceding node having the same anchor. It is an error to have an alias use an anchor that does not occur previously in the serialization of the document. It is not an error to specify an anchor that is not used by any alias node. Note that an alias node may not specify any properties or content, as these were already specified at the first occurrence of the node.

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

Example 4.30. Alias Node

```
---
anchor : &A001 Anchored scalar
override : &A001 The alias node below
  is a repeated use of this text.
alias : *A001
...
# Legend:
#   c-ns-alias-node
#   not alias node
```

```
---
!map {
  !str "anchor":
    &B002 !str "Anchored scalar",
  !str "override":
    &C003 !str "The alias node
      below is a repeated use of this
      text.",
  !str alias: *C003,
}
...
```

Complete Node

A complete node consists of the node properties followed by the node's content. Both the properties and the content are optional. A node with empty content is considered to be in the plain scalar style. To prevent ambiguities, a node used as a a simple key may not be completely empty. An alias node is considered to be a second occurrence of the complete node (both properties and content). Inside a flow collection, only flow node styles may be used. In contrast, block collections may contain both block style nodes and flow style nodes followed by a line break (with an optional comment).

[120]
```
s-l-empty-node ::= /* implicit empty plain content */
                   ( s-discarded-spaces
                     c-nb-throwaway-text? )?
                   b-ignored-any
                   l-comment*
```

Example 4.31. Completely Empty Node

```
---
completely empty:↓
with spaces: ↓
with comments: # empty plain↓
  # scalar value↓
...
# Legend:
#   s-l-empty-node
```

```
---
!map {
  !str "completely empty":
    !str "",
  !str "with spaces":
    !str "",
  !str "with comment":
    !str "",
}
...
```

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

Example 4.32. Flow Node in Flow Context

```
---
{
  content: "foo",
  anchored: &a0 # for alias
    bar,
  alias: *a0,
  empty: # implicit
    !str # plain content
}
...
# Legend:
#   ns-flow-node
```

```
---
!map {
  !str "content":
    !str "foo",
  !str "anchored":
    &a1 !str "bar",
  !str "alias":
    *anchor,
  !str "empty":
    !str ""
}
...
```

```
[122]     ns-l-flow-in-block(n,c) ::= ns-flow-node(n,flow-out)
                                      s-b-seperated-comment
                                      l-comment*
```

Example 4.33. Flow Node in Block Context

```
---
content: "foo"↓
anchored: &a0 # for alias
    bar↓
alias: *a0↓
empty: # implicit
  !str # plain content↓
in seq:
- "content"↓
- !str # implicit plain content↓
...
# Legend:
#   ns-l-flow-in-block
#   not flow-in-block
# TODO: [ a, b ]#ERROR no space
```

```
---
!map {
  !str "content":
    !str "foo",
  !str "anchored":
    &a1 !str "bar",
  !str "alias":
    *anchor,
  !str "empty":
    !str "",
  !str "seq":
    !seq [
      !str "content",
      !str "",
    ]
}
...
```

```
[123]    ns-l-block-in-block(n,c) ::= ( c-ns-properties(n,c)
                                        s-seperate-spaces(n,c) )?
                                      c-l-block-content(n,c)
```

```
[124]      ns-l-block-node(n,c) ::=  ns-l-block-in-block(n,c)
                                   | ns-l-flow-in-block(n,c)
```

Example 4.34. Block Node in Block Context

```
---
folded: !str
    >-
  Word-wrapped
  text lines.↓
↓
literal: |
  #!/usr/bin/perl
  print "Source code\n";↓
sequence:↓
- One
- Two↓
mapping: # Key order need
  !car   # not be preserved.
    make: Ferrari
    color: red
    price: If you have to ask...↓
...
# Legend:
#   ns-l-block-in-block
```

```
---
!map {
  !str "folded":
    !str "Word-wrapped text lines.",
  !str "literal":
    !str "#!/usr/bin/perl\n\
         print \"Source code\\n\";\n",
  !str "sequence":
    !seq [
      !str "One",
      !str "Two",
    ]
  !str "mapping":
    !car {
      !str "make":
        !str "Ferrari",
      !str "color":
        !str "red",
      !str "price":
        !str "If you have to ask..."
    }
}
...
```

Scalar Content

YAML provides three flow styles and two block styles to choose from for presenting scalar contents, depending upon the readability requirements. Throwaway comments may precede or follow scalar content, but may not appear inside it. The comment lines following plain or block scalar content must be less indented than the content itself. Empty lines in a scalar content that are followed by a non-empty content line are interpreted as content rather than as implicit comments. Such lines may be less indented than the non-empty content lines.

Flow Scalar Styles

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

Double Quoted

The double quoted flow scalar style is indicated by surrounding "**`"`**" characters. This is the only style capable of expressing arbitrary Unicode strings, at the cost of some verbosity: escaping line breaks and the printable "**`\`**" and "**`"`**" characters. It is an error for double quoted content to contain invalid escape sequences. If a line break is escaped it is excluded from the content, while white space preceding the break is preserved. This allows double quoted content to be broken at arbitrary positions.

```
[125]          nb-double-char ::=  ( nb-char - “\” - “"” )
                                 | ns-esc-sequence
[126]          ns-double-char ::= nb-double-char - s-char
[127]       nb-ns-double-chunk ::= ns-double-char nb-double-char*
[128]       nb-ns-double-chunk ::= nb-double-char* ns-double-char
```

Example 4.35. Double Quoted Characters

```
---
"T\"'"
...
# Legend:
#   ns-double-char
#   ns-esc-sequence
```

```
---
!str "T\"'"
...
```

```
[129]     s-l-double-folded(n) ::= s-discarded-spaces?
                                   b-l-folded-break(n,flow)
[130]    s-l-double-escaped(n) ::= c-b-escaped l-empty-line(n,flow)*
```

Example 4.36. Double Quoted Line Breaks

```
---
 "   ↓
   ↓
  A   \↓
   B⇓
  C \⇓
 ↓
  "
...
# Legend:
#  s-l-double-folded
#  b-l-double-escaped
```

```
---
!str "\nA  B\LC\n"
...
```

```
[131]      nb-l-double-first(n) ::= nb-ns-double-chunk?
                                    ( s-l-double-folded(n)
                                    | ( s-char*
                                        s-l-double-escaped(n) ) )
[132]     ns-l-double-folded(n) ::= ns-double-char
                                    nb-ns-double-chunk?
                                    s-l-double-folded(n)
[133]    ns-l-double-escaped(n) ::= nb-ns-double-chunk?
                                    s-l-double-escaped(n)
[134]      ns-l-double-inner(n) ::= s-indentation(n) s-discarded-spaces?
                                    ( ns-l-double-folded(n)
                                    | ns-l-double-escaped(n) )
[135]       i-nb-double-last(n) ::= s-indentation(n) s-discarded-spaces?
                                    nb-ns-double-chunk?
```

```
[136]          nb-double-multi(n) ::= nb-l-double-first(n)
                                      ( ns-l-double-inner(n)
                                      - c-forbidden-line )*
                                      i-nb-double-last(n)
```

Example 4.37. Multi Line Double Quoted Scalars

```
---
- "Some↓
↓
quoted text"
- "Some \⇓
quoted text↓
      ↓
"
- "\↓
Some↓
quoted\↓
\ text"
...
# Legend:
# nb-l-double-first
# l-double-inner
# i-nb-double-last
```

```
---
!seq [
  !str "Some\nquoted text",
  !str "Some quoted text\n",
  !str "Some quoted text",
]
...
```

Example 4.38. Invalid Double Quoted Inner Lines

```
"
---↓
...  Text
"
```

```
ERROR:
  Unindented content lines must not
  begin with a document start --- or
  end ... indication.
```

```
[137]         nb-double-single(n) ::= nb-double-char*
[138]            nb-double-any(n) ::=   nb-double-single(n)
                                      | nb-double-multi(n)
[139]         nb-double-text(n,c) ::= c = flow-out ⇒ nb-double-any(n)
                                      c = flow-in  ⇒ nb-double-any(n)
                                      c = flow-key ⇒ nb-double-single(n)
[140]       c-double-quoted(n,c) ::= “"” nb-double-text(n,c) “"”
```

Example 4.39. One Line Double Quoted Scalars

```
---
"key\n" :
  " value "
...
# Legend:
# nb-double-single
```

```
---
!map {
  !str "key\n":
    !str " value "
}
...
```

Single Quoted

The single quoted flow scalar style is indicated by surrounding “'” characters. Therefore, within a single quoted scalar such characters need to be repeated. No other form of escaping is done. In particular, the “\” and “"” characters may be freely used. This restricts single quoted scalars to printable characters.

```
[141]         ns-quoted-quote ::= “'” “'”
[142]          nb-single-char ::=   ( nb-char - “"” )
                                  | ns-quoted-quote
[143]          ns-single-char ::= nb-single-char - s-char
[144]      nb-ns-single-chunk ::= ns-single-char nb-single-char*
[145]      nb-ns-single-chunk ::= nb-single-char* ns-single-char
```

Example 4.40. Single Quoted Characters

```
---
'T\"'''
...
# Legend:
#   ns-single-char
#   ns-quoted-quote
```

```
---
!str "T\\"'"
...
```

```
[146]    nb-l-single-first(n) ::= nb-ns-single-chunk?
                                  s-discarded-spaces?
                                  b-l-folded-break(n,flow)
[147]       l-single-inner(n) ::= s-indentation(n)
                                  s-discarded-spaces?
                                  ( ns-single-char
                                    nb-ns-single-chunk?
                                    s-discarded-spaces? )?
                                  b-l-folded-break(n,flow)
[148]     i-nb-single-last(n) ::= s-indentation(n)
                                  s-discarded-spaces?
                                  nb-ns-single-chunk?
[149]      nb-single-multi(n) ::= nb-l-single-first(n)
                                  ( l-single-inner(n)
                                  - c-forbidden-line )*
                                  i-nb-single-last(n)
```

Example 4.41. Multi Line Single Quoted Scalars

```
---
- 'Some↓
↓
quoted text'
- 'Some⇓
quoted text↓
      ↓
'
- '\↓
Some↓
quoted\↓
\ text'
...
# Legend:
#  nb-l-single-first
#  l-single-inner
#  i-nb-single-last
```

```
---
!seq [
  !str "Some\nquoted text",
  !str "Some\Lquoted text\n",
  !str "\\ Some quoted\\ \\ text",
]
...
```

Example 4.42. Invalid Single Quoted Inner Lines

```
'
---↓
...  Text
'
```

```
ERROR:
  Unindented content lines must not
  begin with a document start --- or
  end ... indication.
```

```
[150]        nb-single-single(n) ::= nb-single-char*
[151]           nb-single-any(n) ::=   nb-single-single(n)
                                     | nb-single-multi(n)
[152]        nb-single-text(n,c) ::= c = flow-out ⇒ nb-single-any(n)
                                     c = flow-in  ⇒ nb-single-any(n)
                                     c = flow-key ⇒ nb-single-single(n)
[153]       c-single-quoted(n,c) ::= “'” nb-single-text(n,c) “'”
```

Example 4.43. One Line Single Quoted Scalars

```
---
'"key\n"' :
  ' value '
...
# Legend:
#  nb-single-single
```

```
---
!map {
  !str "\"key\\n\"":
    !str " value "
}
...
```

Plain

The plain flow scalar style uses no identifying markers, and is therefore the most readable, and most limited, flow scalar style. The exact restrictions on plain content depend on the parsing context. There are three different such contexts. Plain content outside a flow collection (flow-out context) is the least restricted plain scalar format. While it can't start or end with line break or white space characters, or start with most indicators, it may contain any indicator except "**#**" and "**:** ". Plain scalars used inside flow collections (flow-in context) are further restricted not to contain flow indicators. Finally, plain simple keys (flow-key context) are further restricted to a single line. When a node has no explicit content specified, it is assumed to be written in the plain style with the empty string as its content.

```
[154]           nb-plain-char(c) ::= c = flow-out ⇒ nb-plain-char-out
                                     c = flow-in  ⇒ nb-plain-char-key
                                     c = flow-key ⇒ nb-plain-char-key
[155]          nb-plain-char-out ::=   ( nb-char - “:” - “#” )
                                     | ( ns-plain-char(flow-out) “#” )
                                     | ( “:” ns-plain-char(flow-out) )
[156]          nb-plain-char-key ::=   ( nb-char - “:” - “#” - “,”
                                       - “[” - “]” - “{” - “}” )
                                     | ( ns-plain-char(flow-key) “#” )
                                     | ( “:” ns-plain-char(flow-key) )
                                     | ( “,” ns-plain-char(flow-key) )
[157]           ns-plain-char(c) ::= nb-plain-char(c) - s-char
[158]     ns-plain-first-char(c) ::=   ( ns-plain-char(c)
                                       - c-top-indicators )
                                     | ( ( “-” | “?” | “:” | “,” )
                                         ns-plain-char(c) )
[159]       nb-ns-plain-chunk(c) ::= nb-plain-char(c)* ns-plain-char(c)
```

Example 4.44. Plain Characters

```
---
- T"!
- ?T\'
- # Implicit empty plain
...
# Legend:
#   ns-plain-first-char
#   ns-plain-char
```

```
---
!seq [
  !str "T\"!",
  !str "?T\\'",
  !str ""
]
...
```

Example 4.45. Invalid Plain Characters

```
---
first characters: [
  -'17', 'text, - text
]
outside flow:
  no  #
  no :
in flow or key: [
  no  # ,  :
  no [ ] { }
]
...
```

```
ERROR:
  Plain scalars must not start with
  certain indicators and must not
  contain certain characters or
  character combinations.
```

```
[160]         ns-plain-single(c) ::= ns-plain-first-char(c)
                                     nb-ns-plain-chunk(c)?
[161]       s-ns-plain-next(n,c) ::= s-discarded-spaces?
                                     b-l-folded-break(n,flow)+
                                     s-indentation(n)
                                     s-discarded-spaces?
                                     ns-plain-char(c)
                                     nb-ns-plain-chunk(c)?
[162]        ns-plain-multi(n,c) ::= ns-plain-single(c)
                                     s-ns-plain-next(n,c)*
[163]              ns-plain(n,c) ::= c = flow-out ⇒ ns-plain-multi(n,c)?
                                     c = flow-in  ⇒ ns-plain-multi(n,c)?
                                     c = flow-key ⇒ ns-plain-single(c)
```

Example 4.46. Plain Scalars

```
---
{
  key  :
   some   ↓
  ↓
    text ⇓
     value
}
...
# Legend:
#   ns-plain-single
#   s-ns-plain-next
```

```
---
!map {
  !str "key":
    !str "some\ntext\Lvalue"
}
...
```

Block Scalar Styles

Block Header

The presentation of a block scalar content is specified by several indicators given in a header preceding the content itself:

Style Indicator — Block scalars have two possible styles, indicated by "**|**" for literal or "**>**" for folded.

```
[164]      c-style-indicator(s) ::= s = literal ⇒ "|"
                                    s = folded  ⇒ ">"
```

Indentation Indicator — Typically, the indentation level of a block scalar content is detected from its first non-empty content line. This detection fails when this line contains leading white space characters (note it may safely start with a "**#**" character). When detection fails, YAML requires that the indentation level for the content be given explicitly. This level is specified as the integer number of the additional indentation spaces used for the content. If the block scalar begins with lines containing only spaces, and no explicit indentation is given, the processor assumes such lines are empty lines. It is an error for any such leading empty line to contain more spaces than the indentation level that is deduced from the first non-empty content line. It is always valid to specify an explicit indentation level for a block scalar node, though a YAML processor should only do so in cases where detection fails.

```
[165]     c-indent-indicator(m) ::= explicit(m) ⇒ ns-decimal-digit
                                    detect(m)   ⇒ /* empty */
```

Chomp Indicator — By default, all block scalars are clipped. An explicit indicator is required in order to strip the final generic line break or keep all the trailing empty lines as part of the content.

```
[166]      c-chomp-indicator(t) ::= t = strip ⇒ "-"
                                    t = clip  ⇒ /* empty */
                                    t = keep  ⇒ "+"
```

Each block scalar content is prefixed by a header consisting of the above indicators, followed by an ignored line break (with an optional comment). The content is is ended by a less-indented line, the end of the characters stream, or by a document end or start line.

```
[167]   c-b-block-header(s,m,t) ::= c-style-indicator(s)
                                    ( ( c-indent-indicator(m)
                                        c-chomp-indicator(t) )
                                    | ( c-chomp-indicator(t)
                                        c-indent-indicator(m) ) )
                                    s-b-seperated-comment
```

Example 4.47. Block Headers

```
---
- |
- >+
- |2
- |2-
- >+2
...
# Legend:
#   c-b-block-header
```

```
---
!seq {
  !str "",
  !str "",
  !str "",
  !str "",
  !str "",
}
...
```

Example 4.48. Invalid Block Headers

```
---
- +|
- 2>
- 2|+
- +2>
...
```

```
ERROR:
  The style indicator must be
  the first header character.
```

Literal

The literal block scalar style is the simplest scalar style. No processing is performed on literal characters aside from line break normalization, stripping away the indentation, and chomping. The indentation is detected from the first non-empty content line. Explicit indentation must be specified in case this yields the wrong result. Chomping must also be specified unless the default keep processing is used. Since escaping is not done, the literal style is restricted to printable characters and long lines cannot be wrapped. In exchange for these restrictions, literal content offers the most readable presentation for source code or other text containing with significant use of indicators, quotes, escape sequences, and line breaks. In particular, literal content lines may begin with a "#" character.

```
[168]      i-nb-literal-text(n) ::=  ( s-indentation(n) nb-char+ )
                                    - c-forbidden-line
[169]         l-literal-line(n) ::= l-empty-line(n,block)*
                                    i-nb-literal-text(n)
[170]       l-literal-text(n,t) ::= ( ( l-literal-line(n)
                                        b-normalized )*
                                      l-literal-line(n)
                                      b-chomped-break(t) )?
                                    l-trail-chomped(n,t)?
[171]            c-l-literal(n) ::= c-b-block-header(literal,m,t)
                                    l-literal-text(n+m,t)
```

Example 4.49. Literal Scalars

```
---
# Clipped
- |
Literal↓
↓
text↓
⇓
# Indented
- |2
Literal↓
↓
text↓
⇓
# Kept
- |+
Literal↓
↓
text↓
⇓
# Stripped, indented
- |2- # Could be |-2
# Literal↓
↓
text↓
⇓
...
# Legend:
#   c-l-literal
#   s-indentation(n)
#   c-b-throwaway-comment
```

```
---
!seq {
  !str "Literal\n\n text\n",
  !str " Literal\n\ntext\n",
  !str "Literal\n \n text\n\L",
  !str "# Literal\n\ntext"
}
...
```

Example 4.50. Invalid Literal Scalars

```
--- |
# Content, not comment
More content
... Oops!
...
```

```
ERROR:
  Unindented content lines must not
  begin with a document start --- or
  end ... indication.
```

Folded

The folded block scalar style is similar to the literal style. However, unlike a literal content, folded content is subject to block line folding. This allows long lines to be broken anywhere a space character (**`#x20`**) appears, at the cost of requiring an empty line to represent each line feed character.

```
[172]        i-nb-folded-text(n) ::=  ( s-indentation(n) ns-char nb-char* )
                                     - c-forbidden-line
[173]       l-nb-folded-lines(n) ::= l-empty-line(n,block)*
                                     ( i-nb-folded-text(n)
                                       b-l-folded-break(n,block) )*
                                     i-nb-folded-text(n)
[174]  l-nb-start-with-folded(n) ::= l-nb-folded-lines(n)
                                     ( b-normalized
                                       l-nb-start-with-spaced(n) )?
```

Example 4.51. Folded Scalar Folded Lines

```
--- >
 ↓
  Folded↓
  text↓
  ↓
  lines↓
    Indented↓
    lines↓
 Folded↓
  text↓
↓
  lines↓
...
# Legend:
#   l-nb-folded-lines
```

```
---
!str "\n\
      Folded text\n\
      lines\n\
      \  Indented\n\
      \  lines\n\
      Folded text\n\
      lines\n"
...
```

```
[175]      i-nb-indented-text(n) ::= s-indentation(n) s-char nb-char*
[176]     l-nb-indented-lines(n) ::= l-empty-line(n,block)*
                                     ( i-nb-indented-text(n)
                                       b-normalized
                                       l-empty-line(n,block)* )
                                     i-nb-indented-text(n)
[177]  l-nb-start-with-spaced(n) ::= l-nb-indented-lines(n)
                                     ( b-normalized
                                       l-nb-start-with-folded(n) )?
```

Example 4.52. Folded Scalar Indented Lines

```
--- >2
↓
Indented↓
lines↓
↓
  Folded↓
  lines↓
 ↓
  ↓
...
# Legend:
#   l-nb-indented-lines
#   s-indentation
```

```
---
!str "\n\
      \  Indented\n\
      \  lines\n\
      \n\
      Folded lines\n\
      \  \n"
...
```

```
[178]        l-folded-text(n,t) ::= ( ( l-nb-start-with-folded(n)
                                      | l-nb-start-with-spaced(n) )
                                      b-chomped-break(t) )?
                                    l-trail-chomped(n,t)
[179]             c-l-folded(n) ::= c-b-block-header(folded,m,t)
                                    l-folded-text(n+m,t)
```

Example 4.53. Folded Scalars

```
---
# Clipped
- >
Folded↓
↓
text↓
⇓
# Indented
- >2
Folded↓
↓
text↓
⇓
# Kept
- >+
Folded↓
↓
text↓
⇓
# Stripped, indented
- >2- # Could be >-2
# Folded↓
↓
text↓
⇓
...
# Legend:
#   c-l-folded
#   s-indentation(n)
#   c-b-throwaway-comment
```

```
---
!seq {
  !str "Folded\n\n text\n",
  !str " Folded\n\ntext\n",
  !str "Folded\n \n text\n\L",
  !str "# Folded\ntext"
}
...
```

Collection Content

YAML provides a single flow style and a single block style for each of the two collection kinds (sequence and mapping). In addition, YAML provides several compact syntax forms for improved readability of common special cases.

Sequence Styles

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

Flow Sequences

When presented in a flow style, sequence content is denoted by "**[**" and "**]**" characters. Sequence entries are separated by a "**,**" character. A final "**,**" character may follow the last sequence entry. This does not cause ambiguity since sequence entries may not be completely empty. YAML provides a compact form for the case where a flow sequence entry is a mapping with a single (key: value) pair, and neither the mapping node nor its single key node have any properties specified.

```
[180]                 in-flow(c) ::= c = flow-out ⇒ flow-in
                                     c = flow-in  ⇒ flow-in
                                     c = flow-key ⇒ flow-key
[181]       ns-flow-seq-entry(n,c) ::=   ns-flow-node(n,in-flow(c))
                                     | ns-flow-single-pair(n,in-flow(c))
[182]         c-flow-sequence(n,c) ::= "["
                                     s-seperate-spaces(n,c)?
                                     ( ns-flow-seq-entry(n,c)
                                       s-seperate-spaces(n,c)?
                                       "," s-seperate-spaces(n,c) )*
                                     ( ns-flow-seq-entry(n,c)
                                       s-seperate-spaces(n,c)? )?
                                     "]"
```

Example 4.54. Flow Sequences

```
---
[ one, !int "2", # scalars
  three : 3,     # pairs
  [ four, ]      # nested
]
...
# Legend:
#   ns-flow-seq-entry
#   ns-flow-single-pair(n,c)
```

```
---
!seq {
  !str "one",
  !int "2",
  !map {
    !str "three":
      !int "3"
  },
  !seq [
    !str "four"
  ]
}
...
```

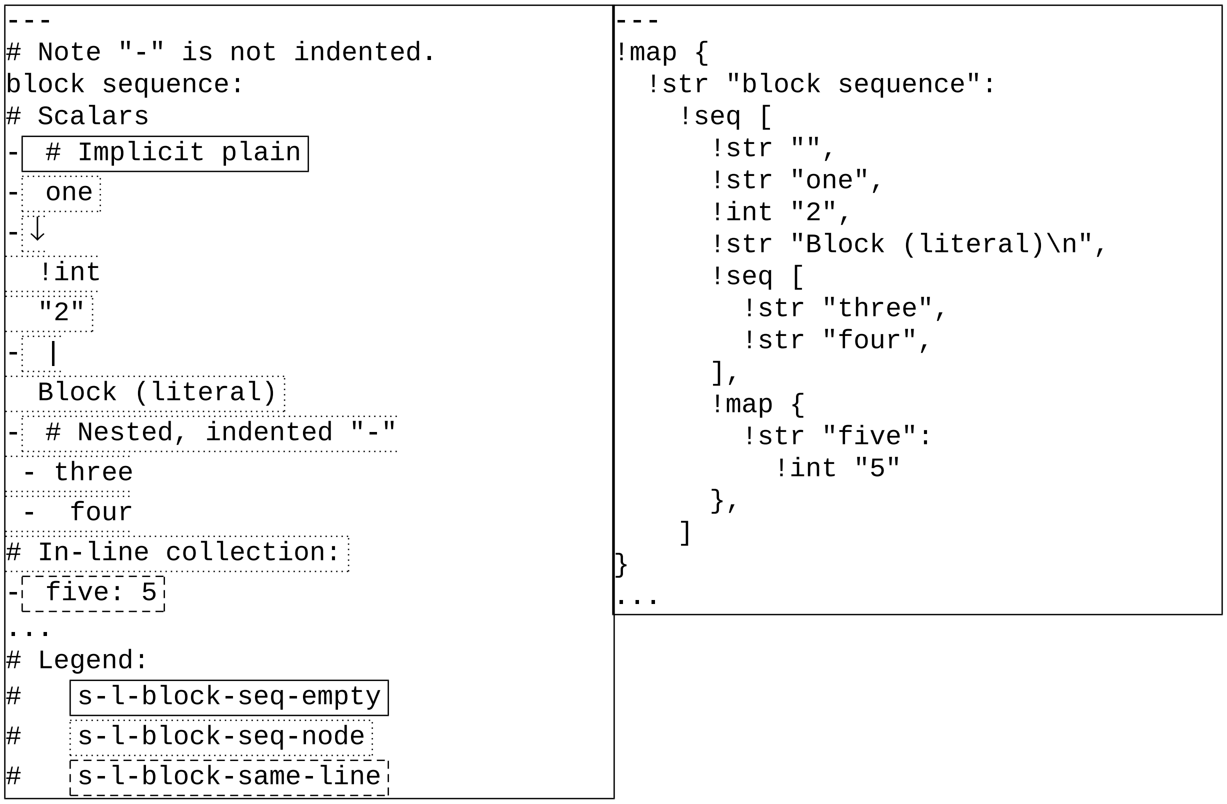
Block Sequences

When presented in the block style, each sequence entry is prefixed by a "**-**" character followed by white space. The "**-**" indicator is considered to be part of the entry's indentation, and hence it need not be more indented than the parent node, unless the parent node is also a block sequence. YAML also provides a compact in-line syntax to be used when the sequence entry is a collection without any node properties specified. In this case both the "**-**" indicator and the spaces following it are considered to be the indentation of the first entry of the contained collection.

```
[183]           seq-spaces(n,c) ::= c = block-out ⇒ n-1
                                    c = block-in  ⇒ n
[184]         s-l-block-seq-empty ::= /* implicit empty plain content */
                                    s-b-seperated-comment
                                    l-comment*
```

```
[185]     s-l-block-seq-node(n,c) ::= s-seperate-spaces(n,c)
                                      ns-l-block-node(n,c)
[186]      s-l-block-same-line(n) ::= s-indentation(m>0)
                                      ( ns-l-in-line-sequence(n+1+m)
                                      | ns-l-in-line-mapping(n+1+m) )
[187]      s-l-block-in-line(n,c) ::=   s-l-block-seq-empty
                                      | s-l-block-seq-node(n,c)
                                      | s-l-block-same-line(n)
[188]       l-block-sequence(n,c) ::= ( s-indentation(seq-spaces(n,c)) “-”
                                        s-l-block-in-line(seq-spaces(n,c),c) )+
```

Example 4.55. Block Sequences



```
---
# Note "-" is not indented.
block sequence:
# Scalars
-  # Implicit plain
-  one
- ↓
  !int
  "2"
-  |
  Block (literal)
-  # Nested, indented "-"
 - three
 -  four
# In-line collection:
-  five: 5
...
# Legend:
#   s-l-block-seq-empty
#   s-l-block-seq-node
#   s-l-block-same-line
```

```
---
!map {
  !str "block sequence":
    !seq [
      !str "",
      !str "one",
      !int "2",
      !str "Block (literal)\n",
      !seq [
        !str "three",
        !str "four",
      ],
      !map {
        !str "five":
          !int "5"
      },
    ]
}
...
```

```
[189]   ns-l-in-line-sequence(n) ::= “-” s-l-block-in-line(n,block-out)
                                     l-block-sequence(n,block-out)?
```

Example 4.56. In-Line Nested Block Sequences

```
---
# In-line nested sequence.
-  - one
- two
  - three
...
# Legend:
#   s-l-block-in-line
#   l-block-sequence
```

```
---
!seq [
  !seq [
    !str "one",
    !str "two",
    !str "three"
  ]
]
...
```

Mapping Styles

The value of a mapping node is an unordered collection of (key: value) pairs. Of necessity, these pairs are presented in some order in the characters stream. This order must not be used in construction of the native data structures. It is an error for two equal key entries to appear in the same mapping value. In such a case the processor may continue, ignoring the second presented key and issuing an appropriate warning. This strategy preserves a consistent information model for streaming and random access applications. Key nodes are denoted by the "**?**" character, and value nodes are denoted by the "**:**" character. For compact presentation, YAML allows omitting the "**:**" indicator for empty plain value nodes that have no properties specified, and omitting the "**?**" indicator for simple keys. This causes potential ambiguity between plain scalars and mapping entries. To resolve this ambiguity without unbound lookahead, simple keys are always written in a flow presentation style, must not be completely empty, are restricted to a single line, and must not span more than a total of 1K (1024) presentation characters. Note that this restricts the total length of (flow) collections used as simple keys. These restrictions are indicated by the use of the **flow-key** production context.

Flow Mappings

When presented in a flow style, a mapping value is denoted by "**{**" and "**}**" characters. Mapping entries are separated by a "**,**" character. A final "**,**" character may follow the last entry. This does not cause ambiguity since mapping entries may not be completely empty. Finally, YAML allows omitting the surrounding "**{**" and "**}**" for a mapping that has no properties specifies, contains a single key: value pair, and is nested inside a flow sequence.

```
[190]    ns-flow-explicit-key(n,c) ::= "?" s-seperate-spaces(n,c)
                                       ns-flow-node(n,in-flow(c))
[191]      ns-flow-simple-key(n,c) ::= ns-flow-node(n,flow-key)
[192]s-ns-flow-explicit-value(n,c) ::= s-seperate-spaces(n,c)?
                                       ":" s-seperate-spaces(n,c)
                                       ns-flow-node(n,in-flow(c))?
[193]  ns-flow-explicit-entry(n,c) ::= ns-flow-explicit-key(n,c)
                                       s-ns-flow-explicit-value(n,c)?
[194]    ns-flow-simple-entry(n,c) ::= ns-flow-simple-key(n,c)
                                       s-ns-flow-explicit-value(n,c)
[195]    ns-flow-implicit-entry(n) ::= ns-flow-simple-key(n,c)
                                       /* implicit empty plain value */
[196]       ns-flow-map-entry(n,c) ::=   ns-flow-explicit-entry(n,c)
                                       | ns-flow-simple-entry(n,c)
                                       | ns-flow-implicit-entry(n)
```

```
[197]         c-flow-mapping(n,c) ::= “{”
                                      s-seperate-spaces(n,c)?
                                      ( ns-flow-map-entry(n,c)
                                        s-seperate-spaces(n,c)?
                                        “,” s-seperate-spaces(n,c) )*
                                      ( ns-flow-map-entry(n,c)
                                        s-seperate-spaces(n,c)? )?
                                      “}”
```

Example 4.57. Flow Mappings

```
---
{ ? explicit
      key : explicit
 value, ? another
 explicit key # implicit value
 , simple key :
  # explicit empty value
 , final simple key # implicit value
}
...
# Legend:
#   ns-flow-explicit-key
#   ns-flow-simple-key
#   s-ns-flow-explicit-value
```

```
---
!map {
  !str "explicit key":
    !str "explicit value",
  !str "another explicit key":
    !str "",
  !str "simple key":
    !str "explicit value",
  !str "final simple key":
    !str ""
}
...
```

```
[198]     ns-flow-single-pair(n,c) ::=   ns-flow-explicit-entry(n,c)
                                       | ns-flow-simple-entry(n,c)
```

Example 4.58. Single Pair Mappings in Flow Sequences

```
---
[ simple key : # explicit empty value
, simple key  : explicit value,
  ? explicit
  key  : explicit value,
  ? explicit
  key  # implicit empty value
]
...
# Legend:
#    ns-flow-explicit-key
#    ns-flow-simple-key
#    s-ns-flow-explicit-value
```

```
---
!seq [
  !map {
    !str "simple key":
      !str "",
  },
  !map {
    !str "simple key":
      !str "explicit value",
  },
  },
  !map {
    !str "explicit key":
      !str "explicit value",
  },
  !map {
    !str "explicit key":
      !str "",
]
...
```

Block Mappings

When presented in a block style, each mapping entry starts on a seperate line. In contrast with block sequences, there is no mandatory indicator character prefixing each entry, as the “**?**” character may be ommitted for simple keys.

```
[199]   ns-l-block-explicit-key(n) ::= “?” s-l-block-in-line(n,block-out)
[200]    l-block-explicit-value(n) ::= s-indentation(n)
                                       “:” s-l-block-in-line(n,block-out)
[201] ns-l-block-explicit-entry(n) ::= ns-l-block-explicit-key(n)
                                       l-block-explicit-value(n)?
```

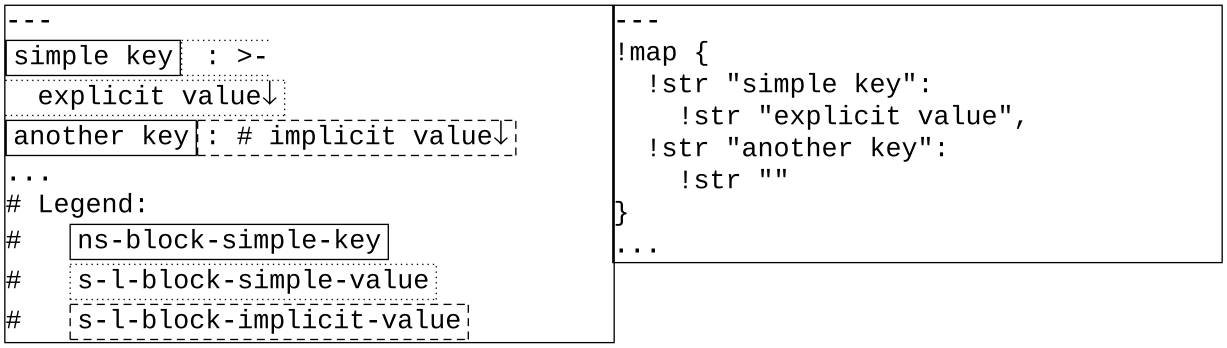
Example 4.59. Explicit Key Block Mapping Entries

```
---
mapping:
  ? >-
   explicit key↓
: explicit value↓
  ? another key
  # implicit empty value
...
# Legend:
#    ns-l-block-explicit-key
#    l-block-explicit-value
```

```
---
!map {
  !str "mapping":
    !map {
      !str "explicit key":
        !str "explicit value",
      !str "another key":
        !str ""
    }
}
...
```

```
[202]      ns-block-simple-key(n) ::= ns-flow-node(n,flow-key)
                                      s-seperate-spaces(n,block-out)? “:”
[203]    s-l-block-simple-value(n) ::= s-seperate-spaces(n,block-out)?
                                      ns-l-block-node(n,block-out)
[204]  s-l-block-implicit-value(n) ::= /* implicit empty plain content */
                                      s-b-seperated-comment
                                      l-comment*
[205]   ns-l-block-simple-entry(n) ::= ns-block-simple-key(n)
                                      ( s-l-block-simple-value(n)
                                      | s-l-block-implicit-value(n) )
```

Example 4.60. Simple Key Block Mapping Entries



```
---
simple key : >-
  explicit value↓
another key: # implicit value↓
...
# Legend:
#   ns-block-simple-key
#   s-l-block-simple-value
#   s-l-block-implicit-value
```

```
---
!map {
  !str "simple key":
    !str "explicit value",
  !str "another key":
    !str ""
}
...
```

```
[206]     ns-l-block-map-entry(n) ::=   ns-l-block-explicit-entry(n)
                                      | ns-l-block-simple-entry(n)
[207]          l-block-mapping(n) ::= ( s-indentation(n)
                                        ns-l-block-map-entry(n) )+
[208]     ns-l-in-line-mapping(n) ::= ns-l-block-map-entry(n)
                                      l-block-mapping(n)?
```

Example 4.61. In-Line Block Mappings

```
---
- one: 1
- mapping:
  ? two: 2
three: 3
    four: 4
  : five: 5
six: 6
    seven: 7
...
# Legend:
#   ns-l-block-map-entry
#   l-block-mapping
```

```
---
!seq {
  !map {
    !str "one":
      !int "1",
  },
  !map {
    !str "mapping":
      !map {
        ? !map {
          !str "two":
            !int "2",
          !str "three":
            !int 3,
          !str "four":
            !int 4,
        }
        : !map {
            !str "five":
              !int "5",
            !str "six":
              !int "6",
            !str "seven":
              !int "7"
          }
      }
  }
}
...
```

YAML Stream

A YAML stream may contain several independent YAML documents. A document header line is used to denote the beginning of a new document. The header line is optional for the first document. A document trailer line may be used to denote the end of a document without starting the next one. Each document is a single node, preceded by optional processor directives. All documents in a single YAML stream must use the same character encoding.

Directive

Directives are instructions to the YAML processor. Like throwaway comments, directives are not reflected in the document's representation graph. Directives apply to a single document. It is an error for the same directive to be specified more than once for the same document.

```
[209]      c-ns-directive ::= “%” ns-directive-name
                              “:” ns-directive-value
[210]   ns-directive-name ::= ( ns-char - “:” )+
[211]  ns-directive-value ::= ns-char+
```

Example 4.62. In-Line Block Mappings

```
--- %YAML:1.0 !map {}
...
# Legend:
#   c-ns-directive
#   ns-directive-name
#   ns-directive-value
```

```
--- %YAML:1.0
!map {
}
...
```

Document Boundaries

Each document may be prefixed by an optional byte order mark to indicate the character encoding, optional comments, and a header line. All documents in a stream must use the same character encoding. If no byte order mark is given for the first document, it is assumed to be in UTF-8 encoding. If no byte order mark is given for a subsequent document, it is assumed to be in the same encoding as the first document. The header line is denoted by "`---`" followed by seperation spaces. YAML allows omitting the the header line of the first document. In this case, the processor must behave as if a header line containing "`---↓`" was specified.

```
[212]     l-document-prefix ::= c-byte-order-mark?
                                l-comment*
[213]      c-document-start ::= "-" "-" "-"
```

Example 4.63. Document Prefix

```
# Prefix before 1st document↓
--- First Document
⇔# Prefix before 2nd document↓
Second Document
# Legend:
#   l-document-prefix
#   c-document-start
#   c-byte-order-mark
```

```
---
!str "First Document"
...
---
!str "Second Document"
...
```

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

```
[214]        c-document-end ::= "." "." "."
[215]     l-document-suffix ::= c-document-end
                                s-b-seperated-comment
                                l-comment*
```

Example 4.64. Document Suffix

```
First Document
... # Suffix following
# the first document.
--- Second Document
... # Suffix following
# the second document
# Legend:
#   l-document-suffix
#   c-document-end
```

```
---
!str "First Document"
...
---
!str "Second Document"
...
```

Since “**`---`**” and “**`...`**” indicate document boundaries, non-indented content lines are forbidden from starting with these character strings, unless they are followed by a non-space character.

```
[216]      c-forbidden-line ::= ( c-document-start
                                | c-document-end )
                                ( s-char | b-any | <EOF> )
```

Documents Stream

A sequence of bytes is a YAML stream if, taken as a whole, it complies with the **`l-yaml-stream`** production. The stream consists of a sequence of disjoint documents. The first document may omit the header line. Note that the stream may contain no documents, even if it contains a non-empty document prefix. A completely empty stream is a valid YAML stream containing no documents.

```
[217]   l-implicit-document ::= l-document-prefix?
                                /* implicit "---↓" header line */
                                s-discarded-spaces?
                                ( c-ns-directive
                                  s-seperate-spaces(0,block-out) )*
                                ns-l-block-node(0,block-out)
                                l-document-suffix?
```

Example 4.65. Shortest Empty Scalar Implicit Document

```
>↓
```

```
---
!str ""
...
```

Example 4.66. Short Empty Scalar Implicit Document

```
""↓
```

```
---
!str ""
...
```

Example 4.67. Longer Empty Scalar Implicit Document

```
!str↓
```

```
---
!str ""
...
```

Example 4.68. Implicit Flow Scalar Document

```
# Implicit documents may be prefixed
  %YAML:1.0 # and may have directives.
    Text # The node itself must not
         # be completely empty.
... # The document suffix
    # is also optional.
```

```
--- %YAML:1.0
!str "Text"
...
```

Example 4.69. Implicit Block Scalar Document

```
>

    This folded text is
    indented by 4 spaces
```

```
---
!str "This folded text is
    indented by 4 spaces"
...
```

Example 4.70. Implicit Flow Collection Document

```
{ key: value }
```

```
---
!map {
  !str "key":
    !str "value"
}
...
```

Example 4.71. Implicit Block Collection Document

```
!seq
- entry
```

```
---
!seq [
  !str "entry"
]
...
```

```
[218]   l-explicit-document ::= l-document-prefix?
                                c-document-start
                                ( s-seperate-spaces(0,block-out)
                                  c-ns-directive )*
                                ( ( s-seperate-spaces(0,block-out)
                                    ns-l-block-node(0,block-out) )
                                | s-l-empty-node )
                                l-document-suffix?
```

Example 4.72. Shortest Explicit Document

```
---↓
```

```
---
!str ""
...
```

Example 4.73. Explicit Flow Scalar Document

```
# The header may be prefixed
# with comments
---
  Text
... # The suffix is
    # also optional.
```

```
---
!str "Text"
...
```

Example 4.74. Explicit Block Scalar Document

```
--- %YAML:1.0 # Directives are optional.
  >
Top level text need
not be indented, and
can't contain a line
starting with --- or
with ...
... # Such a line will
    # end the document.
```

```
--- %YAML:1.0
!str
"Top level text need
not be indented, and
can't contain a line
starting with --- or
with ...\n"
...
```

Example 4.75. Explicit Flow Collection Document

```
--- [ entry ]
```

```
---
!seq [
  !str "entry"
]
...
```

Example 4.76. Explicit Block Collection Document

```
--- !map
key: value
...
```

```
---
!map {
  !str "key":
    !str "value"
}
...
```

```
[219]    l-yaml-stream ::= ( l-document-prefix
                         | ( l-implicit-document?
                             l-explicit-document* ) )
                         <EOF>
```

Example 4.77. Stream Without Document

```
# This stream contains no document.
↓
# All it contains is a prefix.
```

```
↓
# No document here.
↓
```

Example 4.78. Stream With Multiple Documents

```
# HTTP log file:
{ at: 2001-08-12 09:25:00.00 Z,
  type: GET, HTTP: "1.0",
  url: /index.html }
---
at: 2001-08-12 09:25:10.00 Z
type: GET
HTTP: '1.0'
url: '/toc.html'
```

```
---
!map {
  !str "at":
    !timestamp
      "2001-08-12 09:25:00.00 Z"
  !str "type":
    !str "GET",
  !str "HTTP":
    !str "1.0",
  !str "url":
    !str "/index.html"
}
...
---
!map {
  !str "at":
    !timestamp
      "2001-08-12 09:25:10.00 Z"
  !str "type":
    !str "GET",
  !str "HTTP":
    !str "1.0",
  !str "url":
    !str "/toc.html"
}
...
```

Example 4.79. Communication Channel in YAML

```
# A communication channel
# based on a YAML stream:
---
sent at: 2002-06-06 11:46:25.10 Z
payload: Whatever
# Receiver can process this as
# soon as the following is sent:
...
# Even if the next message
# is sent long after:
---
sent at: 2002-06-06 12:05:53.47 Z
payload: Whatever
...
```

```
---
!map {
  !str "sent at":
    !timestamp
      "2002-06-06 11:46:25.10 Z",
  !str "payload":
    !str "Whatever"
}
...
---
!map {
  !str "sent at":
    !timestamp
      "2002-06-06 12:05:53.47 Z"
  !str "payload":
    !str "Whatever"
}
...
```

Appendix A. Tag Repository

Following is a description of the three mandatory core tags. YAML requires support for the seq, map and str tags. YAML also provides a set of universal tags, that are not mandatory, in the YAML tag repository available at http://yaml.org/spec/type.html. These tags represent native data types in most programming languages, or are useful in a wide range of applications. Therefore, applications are strongly encouraged to make use of them whenever they are appropriate, in order to improve interoperability between YAML systems.

Sequence

URI:	**`tag:yaml.org,2002:seq`**
Shorthand:	**`!seq`**
Kind:	Sequence.
Definition:	Collections indexed by sequential integers starting with zero. Example bindings include the Perl array, Python's list or tuple, and Java's array or vector.
Resolution and Validation:	This tag accepts all sequence values. It is is typically used as the fall-back tag for sequence nodes.

Example A.1. `!seq` Examples

```
# The following are equal seqs
# with different identities.
flow: [ one, two ]
spanning: [ one,
     two ]
block:
  - one
  - two
```

Mapping

URI:	**`tag:yaml.org,2002:map`**
Shorthand:	**`!map`**
Kind:	Mapping.
Definition:	Associative container, where each key is unique in the association and mapped to exactly one value. Example bindings include the Perl hash, Python's dictionary, and Java's Hashtable.
Resolution and Validation:	This tag accepts all mapping values. It is is typically used as the fall-back tag for mapping nodes.

Example A.2. `!map` Examples

```
# The following are equal maps
# with different identities.
flow: { one: 1, two: 2 }
block:
    one: 1
    two: 2
```

String

URI:	**`tag:yaml.org,2002:str`**
Shorthand:	**`!str`**
Kind:	Scalar.
Definition:	Unicode strings, a sequence of zero or more Unicode characters. This type is usually bound to the native language's string or character array construct. Note that generic YAML tools should have an immutable (const) interface to such constructs even when the language default is mutable (such as in C/C++).
Canonical Format:	N/A (single format).
Resolution and Validation:	This tag accepts all scalar values. It is is typically used as the fall-back tag for scalar nodes.

Example A.3. `!str` Examples

```
# Assuming an application
# using implicit integers.
- 12     # An integer
# The following scalars
# are loaded to the
# string value '1' '2'.
- !str 12
- '12'
- "12"
- "\
 1\
 2"
# Otherwise, everything is a string:
- /foo/bar
- 192.168.1.1
```

Appendix B. BNF Productions

This appendix repeats all the productions of the syntax in a single continuous production set, for use as a quick reference.

```
[1]                  c-printable ::=   #x9 | #xA | #xD
                                     | [#x20-#x7E] | #x85
                                     | [#xA0-#xD7FF]
                                     | [#xE000-#xFFFD]
                                     | [#x10000-#x10FFFF]
[2]             c-byte-order-mark ::= #xFEFF
[3]              c-sequence-start ::= “[”
[4]                c-sequence-end ::= “]”
[5]               c-mapping-start ::= “{”
[6]                 c-mapping-end ::= “}”
[7]              c-sequence-entry ::= “-”
[8]                 c-mapping-key ::= “?”
[9]               c-mapping-value ::= “:”
[10]              c-collect-entry ::= “,”
[11]                  c-throwaway ::= “#”
[12]                     c-anchor ::= “&”
[13]                      c-alias ::= “*”
[14]                        c-tag ::= “!”
[15]                    c-literal ::= “|”
[16]                     c-folded ::= “>”
[17]               c-single-quote ::= “'”
[18]               c-double-quote ::= “"”
[19]                  c-directive ::= “%”
[20]                   c-reserved ::= “@” | “`”
[21]             c-top-indicators ::=   “[” | “]” | “{” | “}”
                                      | “-” | “?” | “:” | “,”
                                      | “#” | “&” | “*” | “!”
                                      | “|” | “>” | “'” | “"”
                                      | “%” | “@” | “`”
[22]                     c-domain ::= “.”
[23]                       c-date ::= “,”
[24]                       c-path ::= “/”
[25]                     c-prefix ::= “^”
[26]                     c-escape ::= “\”
[27]                      c-strip ::= “-”
[28]                       c-keep ::= “+”
[29]             c-sub-indicators ::=   “.” | “,” | “/” | “^”
                                      | “\” | “-” | “+”
[30]                  b-line-feed ::= #xA /* LF */
[31]            b-carriage-return ::= #xD /* CR */
[32]                       b-next ::= #x85 /* NEL */
[33]             b-line-separator ::= #x2028 /* LS */
[34]        b-paragraph-separator ::= #x2029 /* PS */
```

```
[35]                          b-char ::=   b-line-feed
                                         | b-carriage-return
                                         | b-next
                                         | b-line-separator
                                         | b-paragraph-separator
[36]                       b-generic ::=   ( b-carriage-return b-line-feed )
                                         | b-carriage-return
                                         | b-line-feed
                                         | b-next
[37]                      b-specific ::=   b-line-separator
                                         | b-paragraph-separator
[38]                           b-any ::= b-generic | b-specific
[39]                  b-as-line-feed ::= b-generic
[40]                    b-normalized ::= b-as-line-feed | b-specific
[41]                         nb-char ::= c-printable - b-char
[42]                          s-char ::=   #x9 /* TAB */
                                         | #x20 /* SP */
[43]                         ns-char ::= nb-char - s-char
[44]                 ns-ascii-letter ::=   [#x41-#x5A] /* A-Z */
                                         | [#x61-#x7A] /* a-z */
[45]                ns-decimal-digit ::= [#x30-#x39] /* 0-9 */
[46]                    ns-hex-digit ::=   ns-decimal-digit
                                         | [#x41-#x46] /* A-F */
                                         | [#x61-#x66] /* a-f */
[47]                    ns-word-char ::=   ns-decimal-digit
                                         | ns-ascii-letter | “-”
[48]                     c-b-escaped ::= “\” b-any /* ignored */
[49]                   ns-esc-escape ::= “\” “\”
[50]             ns-esc-double-quote ::= “\” “"”
[51]                      ns-esc-bel ::= “\” “a”
[52]                ns-esc-backspace ::= “\” “b”
[53]                      ns-esc-esc ::= “\” “e”
[54]                ns-esc-form-feed ::= “\” “f”
[55]                ns-esc-line-feed ::= “\” “n”
[56]                   ns-esc-return ::= “\” “r”
[57]                      ns-esc-tab ::= “\” “t”
[58]             ns-esc-vertical-tab ::= “\” “v”
[59]                    ns-esc-caret ::= “\” “^”
[60]                     ns-esc-null ::= “\” “0”
[61]                    ns-esc-space ::= “\” #20 /* SP */
[62]      ns-esc-non-breaking-space ::= “\” “_”
[63]                     ns-esc-next ::= “\” “N”
[64]          ns-esc-line-separator ::= “\” “L”
[65]     ns-esc-paragraph-separator ::= “\” “P”
[66]                    ns-esc-8-bit ::= “\” “x” ( ns-hex-digit x 2 )
[67]                   ns-esc-16-bit ::= “\” “u” ( ns-hex-digit x 4 )
[68]                   ns-esc-32-bit ::= “\” “U” ( ns-hex-digit x 8 )
```

```
[69]              ns-esc-sequence ::=   ns-esc-escape
                                      | ns-esc-double-quote
                                      | ns-esc-bel
                                      | ns-esc-backspace
                                      | ns-esc-esc
                                      | ns-esc-form-feed
                                      | ns-esc-line-feed
                                      | ns-esc-return
                                      | ns-esc-tab
                                      | ns-esc-vertical-tab
                                      | ns-esc-caret
                                      | ns-esc-null
                                      | ns-esc-space
                                      | ns-esc-non-breaking-space
                                      | ns-esc-next
                                      | ns-esc-line-separator
                                      | ns-esc-paragraph-separator
                                      | ns-esc-8-bit
                                      | ns-esc-16-bit
                                      | ns-esc-32-bit
[70]              s-indentation(n) ::= #x20 x n
[71]            s-discarded-spaces ::= s-char+
[72]           c-nb-throwaway-text ::= “#” nb-char*
[73]                 b-ignored-any ::= b-any
[74]        c-b-throwaway-comment ::= c-nb-throwaway-text? b-ignored-any
[75]        s-b-seperated-comment ::= ( s-discarded-spaces
                                        c-nb-throwaway-text? )?
                                      b-ignored-any
[76]            l-empty-comment(n) ::= s-indentation(n)
                                      b-ignored-any
[77]             l-text-comment(n) ::= s-indentation(n)
                                      c-b-throwaway-comment
[78]                     l-comment ::=   l-empty-comment(≥0)
                                      | l-text-comment(≥0)
[79]        s-seperate-spaces(n,c) ::= c = block-out ⇒ s-seperate-span-spaces(n)
                                      c = block-in  ⇒ s-seperate-span-spaces(n)
                                      c = flow-out  ⇒ s-seperate-span-spaces(n)
                                      c = flow-in   ⇒ s-seperate-span-spaces(n)
                                      c = flow-key  ⇒ s-discarded-spaces
[80]    s-seperate-span-spaces(n) ::=   s-discarded-spaces
                                      | ( s-b-seperated-comment
                                          l-comment*
                                          s-indentation(n) s-discarded-spaces?
                                      )
[81]                    b-as-space ::= b-generic
[82]             b-ignored-generic ::= b-generic
[83]      b-l-folded-specific(n,c) ::= b-specific
                                      l-empty-line(n,c)*
[84]      b-l-folded-as-space(n,c) ::= b-as-space
[85]       b-l-folded-trimmed(n,c) ::= b-ignored-generic
                                      l-empty-line(n,c)+
[86]          b-l-folded-break(n,c) ::=   b-l-folded-specific(n,c)
                                      | b-l-folded-as-space(n,c)
                                      | b-l-folded-trimmed(n,c)
[87]            i-s-empty-line(n,c) ::= c = block-out ⇒ i-s-empty-line-block(n)
                                      c = block-in  ⇒ i-s-empty-line-block(n)
                                      c = flow-out  ⇒ i-s-empty-line-flow(n)
                                      c = flow-in   ⇒ i-s-empty-line-flow(n)
[88]        i-s-empty-line-block(n) ::= s-indentation(≤n)
[89]         i-s-empty-line-flow(n) ::=   s-indentation(≤n)
                                      | ( s-indentation(n)
                                          s-discarded-spaces )
[90]          l-empty-specific(n,c) ::= i-s-empty-line(n,c)
                                      b-specific
[91]           l-empty-generic(n,c) ::= i-s-empty-line(n,c)
                                      b-as-line-feed
[92]              l-empty-line(n,c) ::=   l-empty-specific(n,c)
                                      | l-empty-generic(n,c)
[93]             b-chomped-break(t) ::= t = strip ⇒ b-chomped-strip
                                      t = clip  ⇒ b-chomped-keep
                                      t = keep  ⇒ b-chomped-keep
[94]                b-chomped-strip ::= b-ignored-any
[95]                 b-chomped-keep ::= b-normalized
[96]               l-trail-comments ::= l-text-comment(<n) l-comment*
[97]           l-trail-chomped(n,t) ::= t = strip ⇒ l-trail-chomped-strip(n)
                                      t = clip  ⇒ l-trail-chomped-strip(n)
                                      t = keep  ⇒ l-trail-chomped-keep(n)
[98]       l-trail-chomped-strip(n) ::= l-empty-comment(n)*
                                      l-trail-comments?
[99]        l-trail-chomped-keep(n) ::= l-empty-line(n,block)*
                                      l-trail-comments?
[100]            c-ns-anchor-property ::= “&” ns-anchor-name
[101]                  ns-anchor-name ::= ns-char+
[102]                     ns-tag-char ::=   ns-esc-sequence
                                          | ( ns-char - “\” - “^” )
[103]               c-ns-tag-property ::= “!”
                                      ( c-ns-private-tag
                                      | ns-global-tag
                                      | ( Prefix-of-above?
                                          “^”
                                          Suffix-of-above ) )
[104]                c-ns-private-tag ::= “!” ns-tag-char+
[105]                     ns-core-tag ::= ( ns-tag-char - “:” - “/” - “!” )
                                      ( ns-tag-char - “:” - “/” )*
[106]               ns-vocabulary-tag ::= ns-word-char+ “/” ns-tag-char*
[107]                  ns-domain-year ::= ns-decimal-digit x 4
[108]                 ns-domain-month ::= ns-decimal-digit x 2
[109]                   ns-domain-day ::= ns-decimal-digit x 2
[110]                   ns-domain-tag ::= ns-word-char+
                                      ( “.” ns-word-char+ )
                                      “,” ns-domain-year
                                      ( “-” ns-domain-month
                                        ( “-” ns-domain-day )? )?
                                      “/” ns-tag-char*
```

```
[111]                ns-global-tag ::=  ns-core-tag
                                      | ns-vocabulary-tag
                                      | ns-domain-tag
[112]         c-ns-properties(n,c) ::=  ( c-ns-tag-property
                                          ( s-seperate-spaces(n,c)
                                            c-ns-anchor-property )? )
                                      | ( c-ns-anchor-property
                                          ( s-seperate-spaces(n,c)
                                            c-ns-tag-property )? )
[113]          ns-flow-scalar(n,c) ::=  c-double-quoted(n,c)
                                      | c-single-quoted(n,c)
                                      | ns-plain(n,c)
[114]        c-flow-collection(n,c) ::=  c-flow-sequence(n,c)
                                      | c-flow-mapping(n,c)
[115]         ns-flow-content(n,c) ::=  ns-flow-scalar(n,c)
                                      | c-flow-collection(n,c)
[116]          c-l-block-scalar(n) ::=  c-l-folded(n)
                                      | c-l-literal(n)
[117]   c-l-block-collection(n,c) ::= c-b-throwaway-comment
                                      l-comment*
                                      ( l-block-sequence(n,c)
                                      | l-block-mapping(n) )
[118]       c-l-block-content(n,c) ::=  c-l-block-scalar(n)
                                      | c-l-block-collection(n,c)
[119]              c-ns-alias-node ::= “*” ns-anchor-name
[120]               s-l-empty-node ::= /* implicit empty plain content */
                                      ( s-discarded-spaces
                                        c-nb-throwaway-text? )?
                                      b-ignored-any
                                      l-comment*
[121]             ns-flow-node(n,c) ::=  c-ns-alias-node
                                      | ( ( c-ns-properties(n,c)
                                            s-seperate-spaces(n,c) )?
                                          ns-flow-content(n,c) )
                                      | c-ns-properties(n,c)
                                        /* implicit empty plain content */
[122]       ns-l-flow-in-block(n,c) ::= ns-flow-node(n,flow-out)
                                      s-b-seperated-comment
                                      l-comment*
[123]      ns-l-block-in-block(n,c) ::= ( c-ns-properties(n,c)
                                        s-seperate-spaces(n,c) )?
                                      c-l-block-content(n,c)
[124]         ns-l-block-node(n,c) ::=  ns-l-block-in-block(n,c)
                                      | ns-l-flow-in-block(n,c)
[125]               nb-double-char ::=  ( nb-char - “\” - “"” )
                                      | ns-esc-sequence
[126]               ns-double-char ::= nb-double-char - s-char
[127]          nb-ns-double-chunk ::= ns-double-char nb-double-char*
[128]          nb-ns-double-chunk ::= nb-double-char* ns-double-char
[129]         s-l-double-folded(n) ::= s-discarded-spaces?
                                      b-l-folded-break(n,flow)
[130]        s-l-double-escaped(n) ::= c-b-escaped l-empty-line(n,flow)*
```

```
[131]        nb-l-double-first(n) ::= nb-ns-double-chunk?
                                      ( s-l-double-folded(n)
                                      | ( s-char*
                                          s-l-double-escaped(n) ) )
[132]       ns-l-double-folded(n) ::= ns-double-char
                                      nb-ns-double-chunk?
                                      s-l-double-folded(n)
[133]      ns-l-double-escaped(n) ::= nb-ns-double-chunk?
                                      s-l-double-escaped(n)
[134]        ns-l-double-inner(n) ::= s-indentation(n) s-discarded-spaces?
                                      ( ns-l-double-folded(n)
                                      | ns-l-double-escaped(n) )
[135]         i-nb-double-last(n) ::= s-indentation(n) s-discarded-spaces?
                                      nb-ns-double-chunk?
[136]          nb-double-multi(n) ::= nb-l-double-first(n)
                                      ( ns-l-double-inner(n)
                                      - c-forbidden-line )*
                                      i-nb-double-last(n)
[137]         nb-double-single(n) ::= nb-double-char*
[138]            nb-double-any(n) ::=   nb-double-single(n)
                                      | nb-double-multi(n)
[139]         nb-double-text(n,c) ::= c = flow-out ⇒ nb-double-any(n)
                                      c = flow-in  ⇒ nb-double-any(n)
                                      c = flow-key ⇒ nb-double-single(n)
[140]        c-double-quoted(n,c) ::= “"” nb-double-text(n,c) “"”
[141]             ns-quoted-quote ::= “'” “'”
[142]              nb-single-char ::=   ( nb-char - “"” )
                                      | ns-quoted-quote
[143]              ns-single-char ::= nb-single-char - s-char
[144]          nb-ns-single-chunk ::= ns-single-char nb-single-char*
[145]          nb-ns-single-chunk ::= nb-single-char* ns-single-char
[146]        nb-l-single-first(n) ::= nb-ns-single-chunk?
                                      s-discarded-spaces?
                                      b-l-folded-break(n,flow)
[147]           l-single-inner(n) ::= s-indentation(n)
                                      s-discarded-spaces?
                                      ( ns-single-char
                                        nb-ns-single-chunk?
                                        s-discarded-spaces? )?
                                      b-l-folded-break(n,flow)
[148]         i-nb-single-last(n) ::= s-indentation(n)
                                      s-discarded-spaces?
                                      nb-ns-single-chunk?
[149]          nb-single-multi(n) ::= nb-l-single-first(n)
                                      ( l-single-inner(n)
                                      - c-forbidden-line )*
                                      i-nb-single-last(n)
[150]         nb-single-single(n) ::= nb-single-char*
[151]            nb-single-any(n) ::=   nb-single-single(n)
                                      | nb-single-multi(n)
[152]         nb-single-text(n,c) ::= c = flow-out ⇒ nb-single-any(n)
                                      c = flow-in  ⇒ nb-single-any(n)
                                      c = flow-key ⇒ nb-single-single(n)
```

```
[153]        c-single-quoted(n,c) ::= “'” nb-single-text(n,c) “'”
[154]            nb-plain-char(c) ::= c = flow-out ⇒ nb-plain-char-out
                                      c = flow-in  ⇒ nb-plain-char-key
                                      c = flow-key ⇒ nb-plain-char-key
[155]           nb-plain-char-out ::=   ( nb-char - “:” - “#” )
                                      | ( ns-plain-char(flow-out) “#” )
                                      | ( “:” ns-plain-char(flow-out) )
[156]           nb-plain-char-key ::=   ( nb-char - “:” - “#” - “,”
                                        - “[” - “]” - “{” - “}” )
                                      | ( ns-plain-char(flow-key) “#” )
                                      | ( “:” ns-plain-char(flow-key) )
                                      | ( “,” ns-plain-char(flow-key) )
[157]            ns-plain-char(c) ::= nb-plain-char(c) - s-char
[158]      ns-plain-first-char(c) ::=   ( ns-plain-char(c)
                                        - c-top-indicators )
                                      | ( ( “-” | “?” | “:” | “,” )
                                          ns-plain-char(c) )
[159]        nb-ns-plain-chunk(c) ::= nb-plain-char(c)* ns-plain-char(c)
[160]          ns-plain-single(c) ::= ns-plain-first-char(c)
                                      nb-ns-plain-chunk(c)?
[161]        s-ns-plain-next(n,c) ::= s-discarded-spaces?
                                      b-l-folded-break(n,flow)+
                                      s-indentation(n)
                                      s-discarded-spaces?
                                      ns-plain-char(c)
                                      nb-ns-plain-chunk(c)?
[162]         ns-plain-multi(n,c) ::= ns-plain-single(c)
                                      s-ns-plain-next(n,c)*
[163]               ns-plain(n,c) ::= c = flow-out ⇒ ns-plain-multi(n,c)?
                                      c = flow-in  ⇒ ns-plain-multi(n,c)?
                                      c = flow-key ⇒ ns-plain-single(c)
[164]        c-style-indicator(s) ::= s = literal ⇒ “|”
                                      s = folded  ⇒ “>”
[165]       c-indent-indicator(m) ::= explicit(m) ⇒ ns-decimal-digit
                                      detect(m)   ⇒ /* empty */
[166]        c-chomp-indicator(t) ::= t = strip ⇒ “-”
                                      t = clip  ⇒ /* empty */
                                      t = keep  ⇒ “+”
[167]     c-b-block-header(s,m,t) ::= c-style-indicator(s)
                                      ( ( c-indent-indicator(m)
                                          c-chomp-indicator(t) )
                                      | ( c-chomp-indicator(t)
                                          c-indent-indicator(m) ) )
                                      s-b-seperated-comment
[168]        i-nb-literal-text(n) ::=   ( s-indentation(n) nb-char+ )
                                      - c-forbidden-line
[169]           l-literal-line(n) ::= l-empty-line(n,block)*
                                      i-nb-literal-text(n)
[170]         l-literal-text(n,t) ::= ( ( l-literal-line(n)
                                          b-normalized )*
                                        l-literal-line(n)
                                        b-chomped-break(t) )?
                                      l-trail-chomped(n,t)?
```

```
[171]             c-l-literal(n) ::= c-b-block-header(literal,m,t)
                                     l-literal-text(n+m,t)
[172]          i-nb-folded-text(n) ::=   ( s-indentation(n) ns-char nb-char* )
                                     - c-forbidden-line
[173]         l-nb-folded-lines(n) ::= l-empty-line(n,block)*
                                     ( i-nb-folded-text(n)
                                       b-l-folded-break(n,block) )*
                                     i-nb-folded-text(n)
[174]    l-nb-start-with-folded(n) ::= l-nb-folded-lines(n)
                                     ( b-normalized
                                       l-nb-start-with-spaced(n) )?
[175]        i-nb-indented-text(n) ::= s-indentation(n) s-char nb-char*
[176]       l-nb-indented-lines(n) ::= l-empty-line(n,block)*
                                     ( i-nb-indented-text(n)
                                       b-normalized
                                       l-empty-line(n,block)* )
                                     i-nb-indented-text(n)
[177]    l-nb-start-with-spaced(n) ::= l-nb-indented-lines(n)
                                     ( b-normalized
                                       l-nb-start-with-folded(n) )?
[178]           l-folded-text(n,t) ::= ( ( l-nb-start-with-folded(n)
                                       | l-nb-start-with-spaced(n) )
                                       b-chomped-break(t) )?
                                     l-trail-chomped(n,t)
[179]              c-l-folded(n) ::= c-b-block-header(folded,m,t)
                                     l-folded-text(n+m,t)
[180]                 in-flow(c) ::= c = flow-out ⇒ flow-in
                                     c = flow-in  ⇒ flow-in
                                     c = flow-key ⇒ flow-key
[181]       ns-flow-seq-entry(n,c) ::=   ns-flow-node(n,in-flow(c))
                                     | ns-flow-single-pair(n,in-flow(c))
[182]        c-flow-sequence(n,c) ::= “[”
                                     s-seperate-spaces(n,c)?
                                     ( ns-flow-seq-entry(n,c)
                                       s-seperate-spaces(n,c)?
                                       “,” s-seperate-spaces(n,c) )*
                                     ( ns-flow-seq-entry(n,c)
                                       s-seperate-spaces(n,c)? )?
                                     “]”
[183]             seq-spaces(n,c) ::= c = block-out ⇒ n-1
                                     c = block-in  ⇒ n
[184]          s-l-block-seq-empty ::= /* implicit empty plain content */
                                     s-b-seperated-comment
                                     l-comment*
[185]      s-l-block-seq-node(n,c) ::= s-seperate-spaces(n,c)
                                     ns-l-block-node(n,c)
[186]       s-l-block-same-line(n) ::= s-indentation(m>0)
                                     ( ns-l-in-line-sequence(n+1+m)
                                     | ns-l-in-line-mapping(n+1+m) )
[187]       s-l-block-in-line(n,c) ::=   s-l-block-seq-empty
                                     | s-l-block-seq-node(n,c)
                                     | s-l-block-same-line(n)
```

```
[188]       l-block-sequence(n,c) ::= ( s-indentation(seq-spaces(n,c)) “-”
                                        s-l-block-in-line(seq-spaces(n,c),c) )+
[189]    ns-l-in-line-sequence(n) ::= “-” s-l-block-in-line(n,block-out)
                                      l-block-sequence(n,block-out)?
[190]   ns-flow-explicit-key(n,c) ::= “?” s-seperate-spaces(n,c)
                                      ns-flow-node(n,in-flow(c))
[191]     ns-flow-simple-key(n,c) ::= ns-flow-node(n,flow-key)
[192]s-ns-flow-explicit-value(n,c) ::= s-seperate-spaces(n,c)?
                                      “:” s-seperate-spaces(n,c)
                                      ns-flow-node(n,in-flow(c))?
[193] ns-flow-explicit-entry(n,c) ::= ns-flow-explicit-key(n,c)
                                      s-ns-flow-explicit-value(n,c)?
[194]   ns-flow-simple-entry(n,c) ::= ns-flow-simple-key(n,c)
                                      s-ns-flow-explicit-value(n,c)
[195]   ns-flow-implicit-entry(n) ::= ns-flow-simple-key(n,c)
                                      /* implicit empty plain value */
[196]      ns-flow-map-entry(n,c) ::=   ns-flow-explicit-entry(n,c)
                                      | ns-flow-simple-entry(n,c)
                                      | ns-flow-implicit-entry(n)
[197]          c-flow-mapping(n,c) ::= “{”
                                      s-seperate-spaces(n,c)?
                                      ( ns-flow-map-entry(n,c)
                                        s-seperate-spaces(n,c)?
                                        “,” s-seperate-spaces(n,c) )*
                                      ( ns-flow-map-entry(n,c)
                                        s-seperate-spaces(n,c)? )?
                                      “}”
[198]     ns-flow-single-pair(n,c) ::=   ns-flow-explicit-entry(n,c)
                                      | ns-flow-simple-entry(n,c)
[199]  ns-l-block-explicit-key(n) ::= “?” s-l-block-in-line(n,block-out)
[200]   l-block-explicit-value(n) ::= s-indentation(n)
                                      “:” s-l-block-in-line(n,block-out)
[201]ns-l-block-explicit-entry(n) ::= ns-l-block-explicit-key(n)
                                      l-block-explicit-value(n)?
[202]      ns-block-simple-key(n) ::= ns-flow-node(n,flow-key)
                                      s-seperate-spaces(n,block-out)? “:”
[203]   s-l-block-simple-value(n) ::= s-seperate-spaces(n,block-out)?
                                      ns-l-block-node(n,block-out)
[204] s-l-block-implicit-value(n) ::= /* implicit empty plain content */
                                      s-b-seperated-comment
                                      l-comment*
[205]  ns-l-block-simple-entry(n) ::= ns-block-simple-key(n)
                                      ( s-l-block-simple-value(n)
                                      | s-l-block-implicit-value(n) )
[206]     ns-l-block-map-entry(n) ::=   ns-l-block-explicit-entry(n)
                                      | ns-l-block-simple-entry(n)
[207]          l-block-mapping(n) ::= ( s-indentation(n)
                                        ns-l-block-map-entry(n) )+
[208]     ns-l-in-line-mapping(n) ::= ns-l-block-map-entry(n)
                                      l-block-mapping(n)?
[209]              c-ns-directive ::= “%” ns-directive-name
                                      “:” ns-directive-value
[210]           ns-directive-name ::= ( ns-char - “:” )+
[211]       ns-directive-value ::= ns-char+
[212]        l-document-prefix ::= c-byte-order-mark?
                                   l-comment*
[213]         c-document-start ::= “-” “-” “-”
[214]           c-document-end ::= “.” “.” “.”
[215]        l-document-suffix ::= c-document-end
                                   s-b-seperated-comment
                                   l-comment*
[216]         c-forbidden-line ::= ( c-document-start
                                   | c-document-end )
                                   ( s-char | b-any | <EOF> )
[217]      l-implicit-document ::= l-document-prefix?
                                   /* implicit "---↓" header line */
                                   s-discarded-spaces?
                                   ( c-ns-directive
                                     s-seperate-spaces(0,block-out) )*
                                   ns-l-block-node(0,block-out)
                                   l-document-suffix?
[218]      l-explicit-document ::= l-document-prefix?
                                   c-document-start
                                   ( s-seperate-spaces(0,block-out)
                                     c-ns-directive )*
                                   ( ( s-seperate-spaces(0,block-out)
                                       ns-l-block-node(0,block-out) )
                                   | s-l-empty-node )
                                   l-document-suffix?
[219]            l-yaml-stream ::= ( l-document-prefix
                                   | ( l-implicit-document?
                                       l-explicit-document* ) )
                                   <EOF>
```

Appendix C. YAML Terms