
CommonMark Spec

Version 0.29 (2019-04-06)

John MacFarlane

 [Creative Commons BY-SA](https://creativecommons.org/licenses/by-sa/4.0/)

- [1 Introduction](#)
 - [1.1 What is Markdown?](#)
 - [1.2 Why is a spec needed?](#)
 - [1.3 About this document](#)
- [2 Preliminaries](#)
 - [2.1 Characters and lines](#)
 - [2.2 Tabs](#)
 - [2.3 Insecure characters](#)
- [3 Blocks and inlines](#)
 - [3.1 Precedence](#)
 - [3.2 Container blocks and leaf blocks](#)
- [4 Leaf blocks](#)
 - [4.1 Thematic breaks](#)
 - [4.2 ATX headings](#)
 - [4.3 Setext headings](#)
 - [4.4 Indented code blocks](#)
 - [4.5 Fenced code blocks](#)
 - [4.6 HTML blocks](#)
 - [4.7 Link reference definitions](#)
 - [4.8 Paragraphs](#)
 - [4.9 Blank lines](#)
- [5 Container blocks](#)
 - [5.1 Block quotes](#)
 - [5.2 List items](#)
 - [5.3 Lists](#)
- [6 Inlines](#)
 - [6.1 Backslash escapes](#)
 - [6.2 Entity and numeric character references](#)
 - [6.3 Code spans](#)
 - [6.4 Emphasis and strong emphasis](#)
 - [6.5 Links](#)
 - [6.6 Images](#)
 - [6.7 Autolinks](#)
 - [6.8 Raw HTML](#)
 - [6.9 Hard line breaks](#)
 - [6.10 Soft line breaks](#)
 - [6.11 Textual content](#)
- [Appendix: A parsing strategy](#)
 - [Overview](#)
 - [Phase 1: block structure](#)
 - [Phase 2: inline structure](#)

1 Introduction

1.1 What is Markdown?

Markdown is a plain text format for writing structured documents, based on conventions for indicating formatting in email and usenet posts. It was developed by John Gruber (with help from Aaron Swartz) and released in 2004 in the form of a [syntax description](#) and a Perl script (Markdown.pl) for converting Markdown to HTML. In the next decade, dozens of

implementations were developed in many languages. Some extended the original Markdown syntax with conventions for footnotes, tables, and other document elements. Some allowed Markdown documents to be rendered in formats other than HTML. Websites like Reddit, StackOverflow, and GitHub had millions of people using Markdown. And Markdown started to be used beyond the web, to author books, articles, slide shows, letters, and lecture notes.

What distinguishes Markdown from many other lightweight markup syntaxes, which are often easier to write, is its readability. As Gruber writes:

The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. (<http://daringfireball.net/projects/markdown/>)

The point can be illustrated by comparing a sample of [AsciiDoc](#) with an equivalent sample of Markdown. Here is a sample of AsciiDoc from the AsciiDoc manual:

```
1. List item one.
+
List item one continued with a second paragraph followed by an
Indented block.
+
.....
$ ls *.sh
$ mv *.sh ~/tmp
.....
+
List item continued with a third paragraph.

2. List item two continued with an open block.
+
---
This paragraph is part of the preceding list item.

a. This list is nested and does not require explicit item
continuation.
+
This paragraph is part of the preceding list item.

b. List item b.

This paragraph belongs to item two of the outer list.
---
```

And here is the equivalent in Markdown:

```
1. List item one.

List item one continued with a second paragraph followed by an
Indented block.

    $ ls *.sh
    $ mv *.sh ~/tmp

List item continued with a third paragraph.

2. List item two continued with an open block.

This paragraph is part of the preceding list item.

1. This list is nested and does not require explicit item continuation.
```

This paragraph is part of the preceding list item.

2. List item b.

This paragraph belongs to item two of the outer list.

The AsciiDoc version is, arguably, easier to write. You don't need to worry about indentation. But the Markdown version is much easier to read. The nesting of list items is apparent to the eye in the source, not just in the processed document.

1.2 Why is a spec needed?

John Gruber's [canonical description of Markdown's syntax](#) does not specify the syntax unambiguously. Here are some examples of questions it does not answer:

1. How much indentation is needed for a sublist? The spec says that continuation paragraphs need to be indented four spaces, but is not fully explicit about sublists. It is natural to think that they, too, must be indented four spaces, but `Markdown.pl` does not require that. This is hardly a "corner case," and divergences between implementations on this issue often lead to surprises for users in real documents. (See [this comment by John Gruber](#).)
2. Is a blank line needed before a block quote or heading? Most implementations do not require the blank line. However, this can lead to unexpected results in hard-wrapped text, and also to ambiguities in parsing (note that some implementations put the heading inside the blockquote, while others do not). (John Gruber has also spoken [in favor of requiring the blank lines](#).)
3. Is a blank line needed before an indented code block? (`Markdown.pl` requires it, but this is not mentioned in the documentation, and some implementations do not require it.)

```
paragraph
  code?
```

4. What is the exact rule for determining when list items get wrapped in `<p>` tags? Can a list be partially "loose" and partially "tight"? What should we do with a list like this?

1. one
2. two
3. three

Or this?

1. one
 - a
 - b
2. two

(There are some relevant comments by John Gruber [here](#).)

5. Can list markers be indented? Can ordered list markers be right-aligned?

8. item 1
9. item 2
10. item 2a

6. Is this one list with a thematic break in its second item, or two lists separated by a thematic break?

```
* a
* * * * *
* b
```

7. When list markers change from numbers to bullets, do we have two lists or one? (The Markdown syntax description suggests two, but the perl scripts and many other implementations produce one.)

```
1. fee
2. fie
- foe
- fum
```

8. What are the precedence rules for the markers of inline structure? For example, is the following a valid link, or does the code span take precedence ?

```
[a backtick (`)](/url) and [another backtick (`)](/url).
```

9. What are the precedence rules for markers of emphasis and strong emphasis? For example, how should the following be parsed?

```
*foo *bar* baz*
```

10. What are the precedence rules between block-level and inline-level structure? For example, how should the following be parsed?

```
- `a long code span can contain a hyphen like this
- and it can screw things up`
```

11. Can list items include section headings? (Markdown.pl does not allow this, but does allow blockquotes to include headings.)

```
- # Heading
```

12. Can list items be empty?

```
* a
*
* b
```

13. Can link references be defined inside block quotes or list items?

```
> Blockquote [foo].
>
> [foo]: /url
```

14. If there are multiple definitions for the same reference, which takes precedence?

```
[foo]: /url1
[foo]: /url2

[foo][]
```

In the absence of a spec, early implementers consulted `Markdown.pl` to resolve these ambiguities. But `Markdown.pl` was quite buggy, and gave manifestly bad results in many cases, so it was not a satisfactory replacement for a spec.

Because there is no unambiguous spec, implementations have diverged considerably. As a result, users are often surprised to find that a document that renders one way on one system (say, a GitHub wiki) renders differently on another (say, converting to docbook using pandoc). To make matters worse, because nothing in Markdown counts as a “syntax error,” the divergence often isn’t discovered right away.

1.3 About this document

This document attempts to specify Markdown syntax unambiguously. It contains many examples with side-by-side Markdown and HTML. These are intended to double as conformance tests. An accompanying script `spec_tests.py` can be used to run the tests against any Markdown program:

```
python test/spec_tests.py --spec spec.txt --program PROGRAM
```

Since this document describes how Markdown is to be parsed into an abstract syntax tree, it would have made sense to use an abstract representation of the syntax tree instead of HTML. But HTML is capable of representing the structural distinctions we need to make, and the choice of HTML for the tests makes it possible to run the tests against an implementation without writing an abstract syntax tree renderer.

This document is generated from a text file, `spec.txt`, written in Markdown with a small extension for the side-by-side tests. The script `tools/makespec.py` can be used to convert `spec.txt` into HTML or CommonMark (which can then be converted into other formats).

In the examples, the `→` character is used to represent tabs.

2 Preliminaries

2.1 Characters and lines

Any sequence of [characters](#) is a valid CommonMark document.

A [character](#) is a Unicode code point. Although some code points (for example, combining accents) do not correspond to characters in an intuitive sense, all code points count as characters for purposes of this spec.

This spec does not specify an encoding; it thinks of lines as composed of [characters](#) rather than bytes. A conforming parser may be limited to a certain encoding.

A [line](#) is a sequence of zero or more [characters](#) other than newline (U+000A) or carriage return (U+000D), followed by a [line ending](#) or by the end of file.

A [line ending](#) is a newline (U+000A), a carriage return (U+000D) not followed by a newline, or a carriage return and a following newline.

A line containing no characters, or a line containing only spaces (U+0020) or tabs (U+0009), is called a [blank line](#).

The following definitions of character classes will be used in this spec:

A [whitespace character](#) is a space (U+0020), tab (U+0009), newline (U+000A), line tabulation (U+000B), form feed (U+000C), or carriage return (U+000D).

[Whitespace](#) is a sequence of one or more [whitespace characters](#).

A [Unicode whitespace character](#) is any code point in the Unicode *Zs* general category, or a tab (U+0009), carriage return (U+000D), newline (U+000A), or form feed (U+000C).

[Unicode whitespace](#) is a sequence of one or more [Unicode whitespace characters](#).

A [space](#) is U+0020.

A [non-whitespace character](#) is any character that is not a [whitespace character](#).

An [ASCII punctuation character](#) is `!, ", #, $, %, &, ', (,), *, +, ,, -, ., /` (U+0021–2F), `:, ;, <, =, >, ?, @` (U+003A–0040), `[, \,], ^, _`, ``` (U+005B–0060), `{, |, }`, or `~` (U+007B–007E).

A [punctuation character](#) is an [ASCII punctuation character](#) or anything in the general Unicode categories Pc, Pd, Pe, Pf, Pi, Po, or Ps.

2.2 Tabs

Tabs in lines are not expanded to [spaces](#). However, in contexts where whitespace helps to define block structure, tabs behave as if they were replaced by spaces with a tab stop of 4 characters.

Thus, for example, a tab can be used instead of four spaces in an indented code block. (Note, however, that internal tabs are passed through as literal tabs, not expanded to spaces.)

[Example 1](#)

<code>→foo→baz→→bim</code>	<code><pre><code>foo→baz→→bim</code>
	<code></code></pre></code>

[Example 2](#)

<code>••→foo→baz→→bim</code>	<code><pre><code>foo→baz→→bim</code>
	<code></code></pre></code>

[Example 3](#)

<code>•••↵a→a</code>	<code><pre><code>a→a</code>
<code>•••↵→a</code>	<code>↵→a</code>
	<code></code></pre></code>

In the following example, a continuation paragraph of a list item is indented with a tab; this has exactly the same effect as indentation with four spaces would:

[Example 4](#)

<code>••↵foo</code>	<code></code>
	<code></code>
<code>→bar</code>	<code><p>foo</p></code>
	<code><p>bar</p></code>
	<code></code>
	<code></code>

[Example 5](#)

- foo	
	
->->bar	<p>foo</p>
	<pre><code> •bar
	</code></pre>
	
	

Normally the > that begins a block quote may be followed optionally by a space, which is not considered part of the content. In the following case > is followed by a tab, which is treated as if it were expanded into three spaces. Since one of these spaces is considered part of the delimiter, foo is considered to be indented six spaces inside the block quote context, so we get an indented code block starting with two spaces.

Example 6

>->->foo	<blockquote>
	<pre><code> •foo
	</code></pre>
	</blockquote>

Example 7

->->->foo	
	
	<pre><code> •foo
	</code></pre>
	
	

Example 8

...foo	<pre><code>foo
->bar	bar
	</code></pre>

Example 9

• foo	
... bar	foo
-> • baz	
	bar
	
	baz
	
	
	
	
	

Example 10

#>Foo	<h1>Foo</h1>
-------	--------------

Example 11

->->*->	<hr />
-----------	--------

2.3 Insecure characters

For security reasons, the Unicode character U+0000 must be replaced with the REPLACEMENT CHARACTER (U+FFFD).

3 Blocks and inlines

We can think of a document as a sequence of **blocks**—structural elements like paragraphs, block quotations, lists, headings, rules, and code blocks. Some blocks (like block quotes and list items) contain other blocks; others (like headings and paragraphs) contain **inline** content—text, links, emphasized text, images, code spans, and so on.

3.1 Precedence

Indicators of block structure always take precedence over indicators of inline structure. So, for example, the following is a list with two items, not a list with one item containing a code span:

[Example 12](#)

- `one	
- `two`	`one
	two`
	

This means that parsing can proceed in two steps: first, the block structure of the document can be discerned; second, text lines inside paragraphs, headings, and other block constructs can be parsed for inline structure. The second step requires information about link reference definitions that will be available only at the end of the first step. Note that the first step requires processing lines in sequence, but the second can be parallelized, since the inline parsing of one block element does not affect the inline parsing of any other.

3.2 Container blocks and leaf blocks

We can divide blocks into two types: **container blocks**, which can contain other blocks, and **leaf blocks**, which cannot.

4 Leaf blocks

This section describes the different kinds of leaf block that make up a Markdown document.

4.1 Thematic breaks

A line consisting of 0-3 spaces of indentation, followed by a sequence of three or more matching -, _, or * characters, each followed optionally by any number of spaces or tabs, forms a **thematic break**.

[Example 13](#)

***	<hr />
---	<hr />
---	<hr />

Wrong characters:

[Example 14](#)

+++

<p>+++</p>

[Example 15](#)

===

<p>===</p>

Not enough characters:

[Example 16](#)

--

**

--

<p>--

**

--</p>

One to three spaces indent are allowed:

[Example 17](#)

• ***

• • ***

<hr />

<hr />

<hr />

Four spaces is too many:

[Example 18](#)

• • • ***

<pre><code>***

</code></pre>

[Example 19](#)

Foo

• • • ***

<p>Foo

***</p>

More than three characters may be used:

[Example 20](#)

<hr />

Spaces are allowed between the characters:

[Example 21](#)

␣␣␣

<hr />

[Example 22](#)

* * • * * * * *

<hr />

[Example 23](#)

- • • • • • - • • • • • - • • • • • -

<hr />

Spaces are allowed at the end:

[Example 24](#)

-	<hr />
-------------	--------

However, no other characters may occur in the line:

[Example 25](#)

_ . . . a	<p>_ . . . a</p>
a-----	<p>a-----</p>
---a---	<p>---a---</p>

It is required that all of the [non-whitespace characters](#) be the same. So, this is not a thematic break:

[Example 26](#)

-	<p>-</p>
-----	-------------------

Thematic breaks do not need blank lines before or after:

[Example 27](#)

- foo	
***	foo
- bar	
	<hr />
	
	bar
	

Thematic breaks can interrupt a paragraph:

[Example 28](#)

Foo	<p>Foo</p>
***	<hr />
bar	<p>bar</p>

If a line of dashes that meets the above conditions for being a thematic break could also be interpreted as the underline of a [setext heading](#), the interpretation as a [setext heading](#) takes precedence. Thus, for example, this is a setext heading, not a paragraph followed by a thematic break:

[Example 29](#)

Foo	<h2>Foo</h2>
---	<p>bar</p>
bar	

When both a thematic break and a list item are possible interpretations of a line, the thematic break takes precedence:

[Example 30](#)

* Foo	
* * *	Foo
* Bar	
	<hr />
	
	Bar
	

If you want a thematic break in a list item, use a different bullet:

[Example 31](#)

- Foo	
- * * *	Foo
	
	<hr />
	
	

4.2 ATX headings

An [ATX heading](#) consists of a string of characters, parsed as inline content, between an opening sequence of 1–6 unescaped # characters and an optional closing sequence of any number of unescaped # characters. The opening sequence of # characters must be followed by a [space](#) or by the end of line. The optional closing sequence of #s must be preceded by a [space](#) and may be followed by spaces only. The opening # character may be indented 0-3 spaces. The raw contents of the heading are stripped of leading and trailing spaces before being parsed as inline content. The heading level is equal to the number of # characters in the opening sequence.

Simple headings:

[Example 32](#)

# foo	<h1>foo</h1>
## foo	<h2>foo</h2>
### foo	<h3>foo</h3>
#### foo	<h4>foo</h4>
##### foo	<h5>foo</h5>
##### foo	<h6>foo</h6>

More than six # characters is not a heading:

[Example 33](#)

##### foo	<p>##### foo</p>
-----------	------------------

At least one space is required between the # characters and the heading's contents, unless the heading is empty. Note that many implementations currently do not require the space. However, the space was required by the [original ATX implementation](#), and it helps prevent things like the following from being parsed as headings:

[Example 34](#)

#5 bolt	<p>#5 bolt</p>
#hashtag	<p>#hashtag</p>

This is not a heading, because the first # is escaped:

[Example 35](#)

\## foo	<p>## foo</p>
---------	---------------

Contents are parsed as inlines:

[Example 36](#)

# foo *bar* *baz*	<h1>foo bar *baz*</h1>
---------------------	---------------------------------

Leading and trailing [whitespace](#) is ignored in parsing inline content:

[Example 37](#)

#foo	<h1>foo</h1>
------------------	--------------

One to three spaces indentation are allowed:

[Example 38](#)

### foo	<h3>foo</h3>
•## foo	<h2>foo</h2>
••# foo	<h1>foo</h1>

Four spaces are too much:

[Example 39](#)

...# foo	<pre><code># foo </code></pre>
----------	-----------------------------------

[Example 40](#)

foo	<p>foo
...# bar	# bar</p>

A closing sequence of # characters is optional:

[Example 41](#)

## foo ##	<h2>foo</h2>
•### ••bar ...###	<h3>bar</h3>

It need not be the same length as the opening sequence:

[Example 42](#)

# foo #####	<h1>foo</h1>
##### foo ##	<h5>foo</h5>

Spaces are allowed after the closing sequence:

Example 43

### foo ###	<h3>foo</h3>
-------------------	--------------

A sequence of # characters with anything but [spaces](#) following it is not a closing sequence, but counts as part of the contents of the heading:

Example 44

### foo ### b	<h3>foo ### b</h3>
---------------	--------------------

The closing sequence must be preceded by a space:

Example 45

# foo#	<h1>foo#</h1>
--------	---------------

Backslash-escaped # characters do not count as part of the closing sequence:

Example 46

### foo \###	<h3>foo ###</h3>
## foo #\##	<h2>foo ###</h2>
# foo \#	<h1>foo #</h1>

ATX headings need not be separated from surrounding content by blank lines, and they can interrupt paragraphs:

Example 47

****	<hr />
## foo	<h2>foo</h2>
****	<hr />

Example 48

Foo bar	<p>Foo bar</p>
# baz	<h1>baz</h1>
Bar foo	<p>Bar foo</p>

ATX headings can be empty:

Example 49

## .	<h2></h2>
#	<h1></h1>
### ###	<h3></h3>

4.3 Setext headings

A [setext heading](#) consists of one or more lines of text, each containing at least one [non-whitespace character](#), with no more than 3 spaces indentation, followed by a [setext heading underline](#). The lines of text must be such that, were they not followed by the setext heading underline, they would be interpreted as a paragraph: they cannot be interpretable as a [code fence](#), [ATX heading](#), [block quote](#), [thematic break](#), [list item](#), or [HTML block](#).

A [setext heading underline](#) is a sequence of = characters or a sequence of – characters, with no more than 3 spaces indentation and any number of trailing spaces. If a line containing a single – can be interpreted as an empty [list item](#), it should be interpreted this way and not as a [setext heading underline](#).

The heading is a level 1 heading if = characters are used in the [setext heading underline](#), and a level 2 heading if – characters are used. The contents of the heading are the result of parsing the preceding lines of text as CommonMark inline content.

In general, a setext heading need not be preceded or followed by a blank line. However, it cannot interrupt a paragraph, so when a setext heading comes after a paragraph, a blank line is needed between them.

Simple examples:

[Example 50](#)

```
Foo *bar*
=====
Foo *bar*
-----
```

```
<h1>Foo <em>bar</em></h1>
<h2>Foo <em>bar</em></h2>
```

The content of the header may span more than one line:

[Example 51](#)

```
Foo *bar
baz*
=====
```

```
<h1>Foo <em>bar
baz</em></h1>
```

The contents are the result of parsing the headings's raw content as inlines. The heading's raw content is formed by concatenating the lines and removing initial and final [whitespace](#).

[Example 52](#)

```
• Foo *bar
baz*→
=====
```

```
<h1>Foo <em>bar
baz</em></h1>
```

The underlining can be any length:

[Example 53](#)

```
Foo
-----
Foo
=
```

```
<h2>Foo</h2>
<h1>Foo</h1>
```

The heading content can be indented up to three spaces, and need not line up with the underlining:

Example 54

.. Foo	<h2>Foo</h2>
---	<h2>Foo</h2>
	<h1>Foo</h1>
• Foo	

• Foo	
• ==	

Four spaces indent is too much:

Example 55

... Foo	<pre><code>Foo
... ---	---
... Foo	Foo
---	</code></pre>
	<hr />

The setext heading underline can be indented up to three spaces, and may have trailing spaces:

Example 56

Foo	<h2>Foo</h2>
.. ---	

Four spaces is too much:

Example 57

Foo	<p>Foo
... ---	---</p>

The setext heading underline cannot contain internal spaces:

Example 58

Foo	<p>Foo
= =	= </p>
	<p>Foo</p>
Foo	<hr />
--- =	

Trailing spaces in the content line do not cause a line break:

Example 59

Foo ..	<h2>Foo</h2>

Nor does a backslash at the end:

[Example 60](#)

Foo\ ----	<h2>Foo\</h2>
--------------	---------------

Since indicators of block structure take precedence over indicators of inline structure, the following are setext headings:

[Example 61](#)

`Foo ----- ` 	<h2>`Foo</h2> <p>`</p> <h2><a title="a dot</h2> <p>of dashes"/></p>
--	---

The setext heading underline cannot be a [lazy continuation line](#) in a list item or block quote:

[Example 62](#)

> Foo ----	<blockquote> <p>Foo</p> </blockquote> <hr />
---------------	---

[Example 63](#)

> foo bar ===	<blockquote> <p>foo bar ===</p> </blockquote>
---------------------	---

[Example 64](#)

- Foo ----	 Foo <hr />
---------------	---

A blank line is needed between a paragraph and a following setext heading, since otherwise the paragraph becomes part of the heading's content:

[Example 65](#)

Foo Bar ----	<h2>Foo Bar</h2>
--------------------	---------------------

But in general a blank line is not required before or after setext headings:

[Example 66](#)

----	<hr />
Foo	<h2>Foo</h2>
----	<h2>Bar</h2>
Bar	<p>Baz</p>

Baz	

Setext headings cannot be empty:

[Example 67](#)

=====	<p>====</p>
-------	-------------

Setext heading text lines must not be interpretable as block constructs other than paragraphs. So, the line of dashes in these examples gets interpreted as a thematic break:

[Example 68](#)

----	<hr />
----	<hr />

[Example 69](#)

- foo	
-----	foo
	
	<hr />

[Example 70](#)

...foo	<pre><code>foo
----	</code></pre>
	<hr />

[Example 71](#)

> foo	<blockquote>
-----	<p>foo</p>
	</blockquote>
	<hr />

If you want a heading with > foo as its literal text, you can use backslash escapes:

[Example 72](#)

\> foo	<h2>> foo</h2>

Compatibility note: Most existing Markdown implementations do not allow the text of setext headings to span multiple lines. But there is no consensus about how to interpret

```

Foo
bar
----
baz

```

One can find four different interpretations:

1. paragraph “Foo”, heading “bar”, paragraph “baz”
2. paragraph “Foo bar”, thematic break, paragraph “baz”
3. paragraph “Foo bar — baz”
4. heading “Foo bar”, paragraph “baz”

We find interpretation 4 most natural, and interpretation 4 increases the expressive power of CommonMark, by allowing multiline headings. Authors who want interpretation 1 can put a blank line after the first paragraph:

[Example 73](#)

Foo	<code><p>Foo</p></code>
bar	<code><h2>bar</h2></code>
---	<code><p>baz</p></code>
baz	

Authors who want interpretation 2 can put blank lines around the thematic break,

[Example 74](#)

Foo	<code><p>Foo</code>
bar	<code>bar</p></code>
---	<code><hr /></code>
	<code><p>baz</p></code>
baz	

or use a thematic break that cannot count as a [setext heading underline](#), such as

[Example 75](#)

Foo	<code><p>Foo</code>
bar	<code>bar</p></code>
* * *	<code><hr /></code>
baz	<code><p>baz</p></code>

Authors who want interpretation 3 can use backslash escapes:

[Example 76](#)

Foo	<code><p>Foo</code>
bar	<code>bar</code>
\---	<code>---</code>
baz	<code>baz</p></code>

4.4 Indented code blocks

An [indented code block](#) is composed of one or more [indented chunks](#) separated by blank lines. An [indented chunk](#) is a sequence of non-blank lines, each indented four or more spaces. The contents of the code block are the literal contents of the lines, including trailing [line endings](#), minus four spaces of indentation. An indented code block has no [info string](#).

An indented code block cannot interrupt a paragraph, so there must be a blank line between a paragraph and a following indented code block. (A blank line is not needed, however, between

a code block and a following paragraph.)

[Example 77](#)

<pre>...a simpleindented code block</pre>	<pre><pre><code>a simple •indented code block </code></pre></pre>
--	---

If there is any ambiguity between an interpretation of indentation as a code block and as indicating that material belongs to a [list item](#), the list item interpretation takes precedence:

[Example 78](#)

<pre>•• foo ...bar</pre>	<pre> <p>foo</p> <p>bar</p> </pre>
--------------------------	--

[Example 79](#)

<pre>1. •foo ... bar</pre>	<pre> <p>foo</p> bar </pre>
----------------------------	---

The contents of a code block are literal text, and do not get parsed as Markdown:

[Example 80](#)

<pre>...<a/> ...*hi* ... one</pre>	<pre><pre><code>&lt;a/&gt; *hi* - one </code></pre></pre>
--	---

Here we have three chunks separated by blank lines:

[Example 81](#)

<pre>... chunk1 ... chunk2 chunk3</pre>	<pre><pre><code>chunk1 chunk2 chunk3 </code></pre></pre>
---	--

Any initial spaces beyond four will be included in the content, even in interior blank lines:

[Example 82](#)

... chunk1	<pre><code>chunk1
.....	..
..... chunk2	• chunk2
	</code></pre>

An indented code block cannot interrupt a paragraph. (This allows hanging indents and the like.)

[Example 83](#)

Foo	<p>Foo
... bar	bar</p>

However, any non-blank line with fewer than four leading spaces ends the code block immediately. So a paragraph may occur immediately after indented code:

[Example 84](#)

... foo	<pre><code>foo
bar	</code></pre>
	<p>bar</p>

And indented code can occur immediately before and after other kinds of blocks:

[Example 85](#)

# Heading	<h1>Heading</h1>
... foo	<pre><code>foo
Heading	</code></pre>
-----	<h2>Heading</h2>
... foo	<pre><code>foo
-----	</code></pre>
	<hr />

The first line can be indented more than four spaces:

[Example 86](#)

..... foo	<pre><code> ... foo
... bar	bar
	</code></pre>

Blank lines preceding or following an indented code block are not included in it:

[Example 87](#)

.....	<pre><code>foo
... foo	</code></pre>
.....	

Trailing spaces are included in the code block's content:

[Example 88](#)

...foo ..

```
<pre><code>foo ..  
</code></pre>
```

4.5 Fenced code blocks

A [code fence](#) is a sequence of at least three consecutive backtick characters (``) or tildes (~). (Tildes and backticks cannot be mixed.) A [fenced code block](#) begins with a code fence, indented no more than three spaces.

The line with the opening code fence may optionally contain some text following the code fence; this is trimmed of leading and trailing whitespace and called the [info string](#). If the [info string](#) comes after a backtick fence, it may not contain any backtick characters. (The reason for this restriction is that otherwise some inline code would be incorrectly interpreted as the beginning of a fenced code block.)

The content of the code block consists of all subsequent lines, until a closing [code fence](#) of the same type as the code block began with (backticks or tildes), and with at least as many backticks or tildes as the opening code fence. If the leading code fence is indented N spaces, then up to N spaces of indentation are removed from each line of the content (if present). (If a content line is not indented, it is preserved unchanged. If it is indented less than N spaces, all of the indentation is removed.)

The closing code fence may be indented up to three spaces, and may be followed only by spaces, which are ignored. If the end of the containing block (or document) is reached and no closing code fence has been found, the code block contains all of the lines after the opening code fence until the end of the containing block (or document). (An alternative spec would require backtracking in the event that a closing code fence is not found. But this makes parsing much less efficient, and there seems to be no real down side to the behavior described here.)

A fenced code block may interrupt a paragraph, and does not require a blank line either before or after.

The content of a code fence is treated as literal text, not parsed as inlines. The first word of the [info string](#) is typically used to specify the language of the code sample, and rendered in the `class` attribute of the `code` tag. However, this spec does not mandate any particular treatment of the [info string](#).

Here is a simple example with backticks:

[Example 89](#)

```
```\n<\n>\n```
```

```
<pre><code><\n>\n</code></pre>
```

With tildes:

### [Example 90](#)

```
~~~\n<\n>\n~~~
```

```
<pre><code>&lt;\n&gt;\n</code></pre>
```

Fewer than three backticks is not enough:

#### [Example 91](#)

``	<p><code>foo</code></p>
foo	
``	

The closing code fence must use the same character as the opening fence:

#### [Example 92](#)

```	<pre><code>aaa
aaa	~~~
```	</code></pre>
```	

#### [Example 93](#)

~~~	<pre><code>aaa
aaa	~~~
~~~	</code></pre>
~~~	

The closing code fence must be at least as long as the opening fence:

[Example 94](#)

~~~~	<pre><code>aaa
aaa	~~~
~~~~	</code></pre>
~~~~~	

#### [Example 95](#)

~~~~	<pre><code>aaa
aaa	~~~
~~~~	</code></pre>
~~~~	

Unclosed code blocks are closed by the end of the document (or the enclosing [block quote](#) or [list item](#)):

[Example 96](#)

```	<pre><code></code></pre>
-----	--------------------------

#### [Example 97](#)

~~~~~	<pre><code>
~~~	~~~
aaa	aaa
	</code></pre>

#### [Example 98](#)

<pre>&gt; .` `` &gt;  aaa bbb</pre>	<pre>&lt;blockquote&gt; &lt;pre&gt;&lt;code&gt;aaa &lt;/code&gt;&lt;/pre&gt; &lt;/blockquote&gt; &lt;p&gt;bbb&lt;/p&gt;</pre>
-------------------------------------	-------------------------------------------------------------------------------------------------------------------------------

A code block can have all empty lines as its content:

#### [Example 99](#)

<pre>`` ` . . . . `` `</pre>	<pre>&lt;pre&gt;&lt;code&gt; . . &lt;/code&gt;&lt;/pre&gt;</pre>
------------------------------	------------------------------------------------------------------

A code block can be empty:

#### [Example 100](#)

<pre>`` ` `` `</pre>	<pre>&lt;pre&gt;&lt;code&gt;&lt;/code&gt;&lt;/pre&gt;</pre>
----------------------	-------------------------------------------------------------

Fences can be indented. If the opening fence is indented, content lines will have equivalent opening indentation removed, if present:

#### [Example 101](#)

<pre>. ` ``   aaa   aaa   `` `</pre>	<pre>&lt;pre&gt;&lt;code&gt;aaa aaa &lt;/code&gt;&lt;/pre&gt;</pre>
--------------------------------------	---------------------------------------------------------------------

#### [Example 102](#)

<pre>. . ` `` aaa .  aaa aaa . . ` ``</pre>	<pre>&lt;pre&gt;&lt;code&gt;aaa aaa aaa &lt;/code&gt;&lt;/pre&gt;</pre>
---------------------------------------------	-------------------------------------------------------------------------

#### [Example 103](#)

<pre>. . . ` `` . .  aaa . . .  aaa .  aaa . . . ` ``</pre>	<pre>&lt;pre&gt;&lt;code&gt;aaa   aaa aaa &lt;/code&gt;&lt;/pre&gt;</pre>
-------------------------------------------------------------	---------------------------------------------------------------------------

Four spaces indentation produces an indented code block:

#### [Example 104](#)

<pre>. . . . ` `` . . . .  aaa . . . . ` ``</pre>	<pre>&lt;pre&gt;&lt;code&gt;` `` aaa ` ``  &lt;/code&gt;&lt;/pre&gt;</pre>
---------------------------------------------------	----------------------------------------------------------------------------

Closing fences may be indented by 0-3 spaces, and their indentation need not match that of the opening fence:

#### [Example 105](#)

~~~	<pre><code>aaa
aaa	</code></pre>
..	

#### [Example 106](#)

...~~~	<pre><code>aaa
aaa	</code></pre>
..	

This is not a closing fence, because it is indented 4 spaces:

#### [Example 107](#)

~~~	<pre><code>aaa
aaa	...~~~
....	</code></pre>

Code fences (opening and closing) cannot contain internal spaces:

#### [Example 108](#)

~~~ .	<p><code> </code>
aaa	aaa</p>

#### [Example 109](#)

~~~~~	<pre><code>aaa
aaa	~~~.~~~
~~~	</code></pre>

Fenced code blocks can interrupt paragraphs, and can be followed directly by paragraphs, without a blank line between:

#### [Example 110](#)

foo	<p>foo</p>
~~~	<pre><code>bar
bar	</code></pre>
~~~	<p>baz</p>
baz	

Other blocks can also occur before and after fenced code blocks without an intervening blank line:

#### [Example 111](#)



foo	<h2>foo</h2>
---	<pre><code>bar
~~~	</code></pre>
bar	<h1>baz</h1>
~~~	
# baz	

An [info string](#) can be provided after the opening code fence. Although this spec doesn't mandate any particular treatment of the info string, the first word is typically used to specify the language of the code block. In HTML output, the language is normally indicated by adding a class to the `code` element consisting of `language-` followed by the language name.

#### Example 112

```ruby	<pre><code class="language-ruby">
def foo(x)	def foo(x)
•return 3	•return 3
end	end
```	</code></pre>

#### Example 113

~~~~	<pre><code class="language-ruby">
•••ruby startline=3 \$%@#	def foo(x)
def foo(x)	•return 3
•return 3	end
end	</code></pre>
~~~~~	

#### Example 114

````;	<pre><code class="language-; "></code></pre>
````;	

[Info strings](#) for backtick code blocks cannot contain backticks:

#### Example 115

``` aa •`	<p><code>aa</code>
foo	foo</p>

[Info strings](#) for tilde code blocks can contain backticks and tildes:

Example 116

~~~~ aa •` •` ~~~	<pre><code class="language-aa">
foo	foo
~~~~	</code></pre>

Closing code fences cannot have [info strings](#):

Example 117

```	<pre><code>``` aa
``` aa	</code></pre>
```	

## 4.6 HTML blocks

An [HTML block](#) is a group of lines that is treated as raw HTML (and will not be escaped in HTML output).

There are seven kinds of [HTML block](#), which can be defined by their start and end conditions. The block begins with a line that meets a [start condition](#) (after up to three spaces optional indentation). It ends with the first subsequent line that meets a matching [end condition](#), or the last line of the document, or the last line of the [container block](#) containing the current HTML block, if no line is encountered that meets the [end condition]. If the first line meets both the [start condition](#) and the [end condition], the block will contain just that line.

1. **Start condition:** line begins with the string `<script`, `<pre`, or `<style` (case-insensitive), followed by whitespace, the string `>`, or the end of the line.  
**End condition:** line contains an end tag `</script>`, `</pre>`, or `</style>` (case-insensitive; it need not match the start tag).
2. **Start condition:** line begins with the string `<!--`.  
**End condition:** line contains the string `-->`.
3. **Start condition:** line begins with the string `<?`.  
**End condition:** line contains the string `?>`.
4. **Start condition:** line begins with the string `<!` followed by an uppercase ASCII letter.  
**End condition:** line contains the character `>`.
5. **Start condition:** line begins with the string `<![CDATA[`.  
**End condition:** line contains the string `]]>`.
6. **Start condition:** line begins the string `<` or `</` followed by one of the strings (case-insensitive) `address`, `article`, `aside`, `base`, `basefont`, `blockquote`, `body`, `caption`, `center`, `col`, `colgroup`, `dd`, `details`, `dialog`, `dir`, `div`, `dl`, `dt`, `fieldset`, `figcaption`, `figure`, `footer`, `form`, `frame`, `frameset`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `header`, `hr`, `html`, `iframe`, `legend`, `li`, `link`, `main`, `menu`, `menuitem`, `nav`, `noframes`, `ol`, `optgroup`, `option`, `p`, `param`, `section`, `source`, `summary`, `table`, `tbody`, `td`, `tfoot`, `th`, `thead`, `title`, `tr`, `track`, `ul`, followed by [whitespace](#), the end of the line, the string `>`, or the string `/>`.  
**End condition:** line is followed by a [blank line](#).
7. **Start condition:** line begins with a complete [open tag](#) (with any [tag name](#) other than `script`, `style`, or `pre`) or a complete [closing tag](#), followed only by [whitespace](#) or the end of the line.  
**End condition:** line is followed by a [blank line](#).

HTML blocks continue until they are closed by their appropriate [end condition], or the last line of the document or other [container block](#). This means any HTML **within an HTML block** that might otherwise be recognised as a start condition will be ignored by the parser and passed through as-is, without changing the parser's state.

For instance, `<pre>` within a HTML block started by `<table>` will not affect the parser state; as the HTML block was started in by start condition 6, it will end at any blank line. This can be surprising:

[Example 118](#)

```

<table><tr><td>
<pre>
Hello,

world.
</pre>
</td></tr></table>

```

```

<table><tr><td>
<pre>
Hello,
<p>world.
</pre></p>
</td></tr></table>

```

In this case, the HTML block is terminated by the newline — the **Hello** text remains verbatim — and regular parsing resumes, with a paragraph, emphasised world and inline and block HTML following.

All types of [HTML blocks](#) except type 7 may interrupt a paragraph. Blocks of type 7 may not interrupt a paragraph. (This restriction is intended to prevent unwanted interpretation of long tags inside a wrapped paragraph as starting HTML blocks.)

Some simple examples follow. Here are some basic HTML blocks of type 6:

#### [Example 119](#)

```

<table>
•<tr>
...<td>
.....hi
...</td>
•</tr>
</table>

```

okay.

```

<table>
•<tr>
...<td>
.....hi
...</td>
•</tr>
</table>
<p>okay. </p>

```

#### [Example 120](#)

```

<div>
•*hello*
.....<foo><a>

```

```

<div>
•*hello*
.....<foo><a>

```

A block can also start with a closing tag:

#### [Example 121](#)

```

</div>
foo

```

```

</div>
foo

```

Here we have two HTML blocks with a Markdown paragraph between them:

#### [Example 122](#)

```

<DIV CLASS="foo">
Markdown
</DIV>

```

```

<DIV CLASS="foo">
<p>Markdown</p>
</DIV>

```

The tag on the first line can be partial, as long as it is split where there would be whitespace:

#### [Example 123](#)

```
<div id="foo"
 •class="bar">
</div>
```

```
<div id="foo"
 •class="bar">
</div>
```

#### [Example 124](#)

```
<div id="foo" class="bar
 •baz">
</div>
```

```
<div id="foo" class="bar
 •baz">
</div>
```

An open tag need not be closed:

#### [Example 125](#)

```
<div>
foo

bar
```

```
<div>
foo
<p>bar</p>
```

A partial tag need not even be completed (garbage in, garbage out):

#### [Example 126](#)

```
<div id="foo"
hi
```

```
<div id="foo"
hi
```

#### [Example 127](#)

```
<div class
foo
```

```
<div class
foo
```

The initial tag doesn't even need to be a valid tag, as long as it starts like one:

#### [Example 128](#)

```
<div *???-&&&-<---
foo
```

```
<div *???-&&&-<---
foo
```

In type 6 blocks, the initial tag need not be on a line by itself:

#### [Example 129](#)

```
<div>*foo*</div>
```

```
<div>*foo*</div>
```

#### [Example 130](#)

```
<table><tr><td>
foo
</td></tr></table>
```

```
<table><tr><td>
foo
</td></tr></table>
```

Everything until the next blank line or end of document gets included in the HTML block. So, in the following example, what looks like a Markdown code block is actually part of the HTML block, which continues until a blank line or the end of the document is reached:

#### [Example 131](#)

```
<div></div>
int x = 33;
```

```
<div></div>
int x = 33;
```

To start an [HTML block](#) with a tag that is *not* in the list of block-level tags in (6), you must put the tag by itself on the first line (and it must be complete):

#### [Example 132](#)

```

bar

```

```

bar

```

In type 7 blocks, the [tag name](#) can be anything:

#### [Example 133](#)

```
<Warning>
bar
</Warning>
```

```
<Warning>
bar
</Warning>
```

#### [Example 134](#)

```
<i class="foo">
bar
</i>
```

```
<i class="foo">
bar
</i>
```

#### [Example 135](#)

```
</ins>
bar
```

```
</ins>
bar
```

These rules are designed to allow us to work with tags that can function as either block-level or inline-level tags. The `<del>` tag is a nice example. We can surround content with `<del>` tags in three different ways. In this case, we get a raw HTML block, because the `<del>` tag is on a line by itself:

#### [Example 136](#)

```

foo

```

```

foo

```

In this case, we get a raw HTML block that just includes the `<del>` tag (because it ends with the following blank line). So the contents get interpreted as CommonMark:

#### [Example 137](#)

```

foo

```

```

<p>foo</p>

```

Finally, in this case, the `<del>` tags are interpreted as [raw HTML](#) *inside* the CommonMark paragraph. (Because the tag is not on a line by itself, we get inline HTML rather than an [HTML block](#).)

#### [Example 138](#)

```
foo <p>foo</p>
```

HTML tags designed to contain literal content (`script`, `style`, `pre`), comments, processing instructions, and declarations are treated somewhat differently. Instead of ending at the first blank line, these blocks end at the first line containing a corresponding end tag. As a result, these blocks can contain blank lines:

A `pre` tag (type 1):

#### [Example 139](#)

```
<pre language="haskell"><code>
import Text.HTML.TagSoup

main = IO ()
main = print $ parseTags tags
</code></pre>
okay

<pre language="haskell"><code>
import Text.HTML.TagSoup

main = IO ()
main = print $ parseTags tags
</code></pre>
<p>okay</p>
```

A `script` tag (type 1):

#### [Example 140](#)

```
<script type="text/javascript">
// JavaScript example

document.getElementById("demo").innerHTML = •
"Hello JavaScript!";
</script>
okay

<script type="text/javascript">
// JavaScript example

document.getElementById("demo").innerHTML = •
"Hello JavaScript!";
</script>
<p>okay</p>
```

A `style` tag (type 1):

#### [Example 141](#)

```
<style
• type="text/css">
h1 {color:red;}

p {color:blue;}
</style>
okay

<style
• type="text/css">
h1 {color:red;}

p {color:blue;}
</style>
<p>okay</p>
```

If there is no matching end tag, the block will end at the end of the document (or the enclosing [block quote](#) or [list item](#)):

#### [Example 142](#)

```
<style
 •type="text/css">

foo
```

```
<style
 •type="text/css">

foo
```

#### [Example 143](#)

```
> <div>
> foo

bar
```

```
<blockquote>
<div>
foo
</blockquote>
<p>bar</p>
```

#### [Example 144](#)

```
- <div>
- foo
```

```


<div>

foo

```

The end tag can occur on the same line as the start tag:

#### [Example 145](#)

```
<style>p{color:red;}</style>
foo
```

```
<style>p{color:red;}</style>
<p>foo</p>
```

#### [Example 146](#)

```
<!-- foo -->*bar*
baz
```

```
<!-- foo -->*bar*
<p>baz</p>
```

Note that anything on the last line after the end tag will be included in the [HTML block](#):

#### [Example 147](#)

```
<script>
foo
</script>l. *bar*
```

```
<script>
foo
</script>l. *bar*
```

A comment (type 2):

#### [Example 148](#)

```
<!-- Foo

bar
 ••baz -->
okay
```

```
<!-- Foo

bar
 ••baz -->
<p>okay</p>
```

A processing instruction (type 3):

#### [Example 149](#)

```
<?php
 •echo ' ' ;

?>
okay
```

```
<?php
 •echo ' ' ;

?>
<p>okay</p>
```

A declaration (type 4):

#### [Example 150](#)

```
<!DOCTYPE html>
```

```
<!DOCTYPE html>
```

CDATA (type 5):

#### [Example 151](#)

```
<![CDATA[
function matchwo(a,b)
{
 •if (a < b && a < 0) then •{
 •••return 1;

 •} else •{

 •••return 0;
 •}
}
]]>
okay
```

```
<![CDATA[
function matchwo(a,b)
{
 •if (a < b && a < 0) then •{
 •••return 1;

 •} else •{

 •••return 0;
 •}
}
]]>
<p>okay</p>
```

The opening tag can be indented 1-3 spaces, but not 4:

#### [Example 152](#)

```
•<!-- foo -->

...<!-- foo -->
```

```
•<!-- foo -->
<pre><code><!-- foo -->
</code></pre>
```

#### [Example 153](#)

```
•<div>

...<div>
```

```
•<div>
<pre><code><div>
</code></pre>
```

An HTML block of types 1–6 can interrupt a paragraph, and need not be preceded by a blank line.

#### [Example 154](#)

```
Foo
<div>
bar
</div>
```

```
<p>Foo</p>
<div>
bar
</div>
```



However, a following blank line is needed, except at the end of a document, and except for blocks of types 1–5, [above](#):

#### [Example 155](#)

<code>&lt;div&gt;</code>	<code>&lt;div&gt;</code>
<code>bar</code>	<code>bar</code>
<code>&lt;/div&gt;</code>	<code>&lt;/div&gt;</code>
<code>*foo*</code>	<code>*foo*</code>

HTML blocks of type 7 cannot interrupt a paragraph:

#### [Example 156](#)

<code>Foo</code>	<code>&lt;p&gt;Foo</code>
<code>&lt;a href="bar"&gt;</code>	<code>&lt;a href="bar"&gt;</code>
<code>baz</code>	<code>baz&lt;/p&gt;</code>

This rule differs from John Gruber’s original Markdown syntax specification, which says:

The only restrictions are that block-level HTML elements — e.g. `<div>`, `<table>`, `<pre>`, `<p>`, etc. — must be separated from surrounding content by blank lines, and the start and end tags of the block should not be indented with tabs or spaces.

In some ways Gruber’s rule is more restrictive than the one given here:

- It requires that an HTML block be preceded by a blank line.
- It does not allow the start tag to be indented.
- It requires a matching end tag, which it also does not allow to be indented.

Most Markdown implementations (including some of Gruber’s own) do not respect all of these restrictions.

There is one respect, however, in which Gruber’s rule is more liberal than the one given here, since it allows blank lines to occur inside an HTML block. There are two reasons for disallowing them here. First, it removes the need to parse balanced tags, which is expensive and can require backtracking from the end of the document if no matching end tag is found. Second, it provides a very simple and flexible way of including Markdown content inside HTML tags: simply separate the Markdown from the HTML using blank lines:

Compare:

#### [Example 157](#)

<code>&lt;div&gt;</code>	<code>&lt;div&gt;</code>
<code>*Emphasized* ¶text.</code>	<code>&lt;p&gt;&lt;em&gt;Emphasized&lt;/em&gt; ¶text.&lt;/p&gt;</code>
<code>&lt;/div&gt;</code>	<code>&lt;/div&gt;</code>

#### [Example 158](#)

<code>&lt;div&gt;</code>	<code>&lt;div&gt;</code>
<code>*Emphasized* ¶text.</code>	<code>*Emphasized* ¶text.</code>
<code>&lt;/div&gt;</code>	<code>&lt;/div&gt;</code>

Some Markdown implementations have adopted a convention of interpreting content inside tags as text if the open tag has the attribute `markdown=1`. The rule given above seems a simpler and more elegant way of achieving the same expressive power, which is also much simpler to parse.

The main potential drawback is that one can no longer paste HTML blocks into Markdown documents with 100% reliability. However, *in most cases* this will work fine, because the blank lines in HTML are usually followed by HTML block tags. For example:

#### Example 159

<code>&lt;table&gt;</code>	<code>&lt;table&gt;</code>
	<code>&lt;tr&gt;</code>
<code>&lt;tr&gt;</code>	<code>&lt;td&gt;</code>
	Hi
<code>&lt;td&gt;</code>	<code>&lt;/td&gt;</code>
Hi	<code>&lt;/tr&gt;</code>
<code>&lt;/td&gt;</code>	<code>&lt;/table&gt;</code>
<code>&lt;/tr&gt;</code>	
<code>&lt;/table&gt;</code>	

There are problems, however, if the inner tags are indented *and* separated by spaces, as then they will be interpreted as an indented code block:

#### Example 160

<code>&lt;table&gt;</code>	<code>&lt;table&gt;</code>
	<code>•&lt;tr&gt;</code>
<code>•&lt;tr&gt;</code>	<code>&lt;pre&gt;&lt;code&gt;&amp;lt;td&amp;gt;</code>
	<code>•Hi</code>
<code>...&lt;td&gt;</code>	<code>&amp;lt;/td&amp;gt;</code>
<code>.....Hi</code>	<code>&lt;/code&gt;&lt;/pre&gt;</code>
<code>...&lt;/td&gt;</code>	<code>•&lt;/tr&gt;</code>
<code>•&lt;/tr&gt;</code>	<code>&lt;/table&gt;</code>
<code>&lt;/table&gt;</code>	

Fortunately, blank lines are usually not necessary and can be deleted. The exception is inside `<pre>` tags, but as described [above](#), raw HTML blocks starting with `<pre>` *can* contain blank lines.

## 4.7 Link reference definitions

A [link reference definition](#) consists of a [link label](#), indented up to three spaces, followed by a colon (:), optional [whitespace](#) (including up to one [line ending](#)), a [link destination](#), optional [whitespace](#) (including up to one [line ending](#)), and an optional [link title](#), which if it is present must be separated from the [link destination](#) by [whitespace](#). No further [non-whitespace characters](#) may occur on the line.

A [link reference definition](#) does not correspond to a structural element of a document. Instead, it defines a label which can be used in [reference links](#) and reference-style [images](#) elsewhere in the document. [Link reference definitions](#) can come either before or after the links that use them.

#### Example 161

```
[foo]: /url "title"
```

```
[foo]
```

```
<p>foo</p>
```

#### [Example 162](#)

```
..[foo]: .
...../url ..
.....'the title' ..
```

```
[foo]
```

```
<p>foo
</p>
```

#### [Example 163](#)

```
[Foo*bar\]]:my_(url) 'title (with parens)'
```

```
[Foo*bar\]]
```

```
<p><a href="my_(url)" title="title (with
parens)">Foo*bar]</p>
```

#### [Example 164](#)

```
[Foo bar]:
<my url>
'title'
```

```
[Foo bar]
```

```
<p>Foo •
bar</p>
```

The title may extend over multiple lines:

#### [Example 165](#)

```
[foo]: /url 'title
title
line1
line2
,
```

```
[foo]
```

```
<p><a href="/url" title="title
title
line1
line2
>foo</p>
```

However, it may not contain a [blank line](#):

#### [Example 166](#)

```
[foo]: /url 'title
```

```
with blank line'
```

```
[foo]
```

```
<p>[foo]: /url 'title</p>
<p>with blank line'</p>
<p>[foo]</p>
```

The title may be omitted:

#### [Example 167](#)

```
[foo]:
/url
```

```
[foo]
```

```
<p>foo</p>
```

The link destination may not be omitted:

[Example 168](#)

[foo]:	<p>[foo]:</p>
[foo]	<p>[foo]</p>

However, an empty link destination may be specified using angle brackets:

[Example 169](#)

[foo]: <>	<p><a href="">foo</a></p>
[foo]	

The title must be separated from the link destination by whitespace:

[Example 170](#)

[foo]: <bar>(baz)	<p>[foo]: <bar>(baz)</p>
[foo]	<p>[foo]</p>

Both title and destination can contain backslash escapes and literal backslashes:

[Example 171](#)

[foo]: ↵url\bar\*baz "foo\"bar\baz"	<p><a href="/url%5Cbar*baz" •
[foo]	title="foo&quot;bar\baz">foo</a></p>

A link can come before its corresponding definition:

[Example 172](#)

[foo]	<p><a href="url">foo</a></p>
[foo]: ↵url	

If there are several matching definitions, the first one takes precedence:

[Example 173](#)

[foo]	<p><a href="first">foo</a></p>
[foo]: ↵first	
[foo]: ↵second	

As noted in the section on [Links](#), matching of labels is case-insensitive (see [matches](#)).

[Example 174](#)

[F00]: ↗url

<p><a href="/url">Foo</a></p>

[Foo]

#### [Example 175](#)

[Α Γ Ω]: ↗φ ο υ

<p><a href="/%CF%86%CE%BF%CF%85">α γ ω</a>  
</p>

[α γ ω]

Here is a link reference definition with no corresponding link. It contributes nothing to the document.

#### [Example 176](#)

[foo]: ↗url

Here is another one:

#### [Example 177](#)

[  
foo  
]: ↗url  
bar

<p>bar</p>

This is not a link reference definition, because there are [non-whitespace characters](#) after the title:

#### [Example 178](#)

[foo]: ↗url "title" ok

<p>[foo]: ↗url &quot;title&quot; ok</p>

This is a link reference definition, but it has no title:

#### [Example 179](#)

[foo]: ↗url  
"title" ok

<p>&quot;title&quot; ok</p>

This is not a link reference definition, because it is indented four spaces:

#### [Example 180](#)

...[foo]: ↗url "title"  
[foo]

<pre><code>[foo]: ↗url &quot;title&quot;  
</code></pre>  
<p>[foo]</p>

This is not a link reference definition, because it occurs inside a code block:

#### [Example 181](#)

```


[foo]: ↗url


```

[foo]

```

<pre><code>[foo]: ↗url
</code></pre>
<p>[foo]</p>

```

A [link reference definition](#) cannot interrupt a paragraph.

#### Example 182

```

Foo
[bar]: ↗baz

[bar]

```

```

<p>Foo
[bar]: ↗baz</p>
<p>[bar]</p>

```

However, it can directly follow other block elements, such as headings and thematic breaks, and it need not be followed by a blank line.

#### Example 183

```

↗Foo
[foo]: ↗url
> ↗bar

```

```

<h1>Foo</h1>
<blockquote>
<p>bar</p>
</blockquote>

```

#### Example 184

```

[foo]: ↗url
bar
===
[foo]

```

```

<h1>bar</h1>
<p>foo</p>

```

#### Example 185

```

[foo]: ↗url
===
[foo]

```

```

<p>===
foo</p>

```

Several [link reference definitions](#) can occur one after another, without intervening blank lines.

#### Example 186

```

[foo]: ↗foo-url ↗foo
[bar]: ↗bar-url
•↗bar
[baz]: ↗baz-url

[foo],
[bar],
[baz]

```

```

<p>foo,
bar,
baz</p>

```

[Link reference definitions](#) can occur inside block containers, like lists and block quotations. They affect the entire document, not just the container in which they are defined:

#### Example 187

[foo]	<p><a href="/url">foo</a></p>
> [foo]: /url	<blockquote>
	</blockquote>

Whether something is a [link reference definition](#) is independent of whether the link reference it defines is used in the document. Thus, for example, the following document contains just a link reference definition, and no visible content:

#### [Example 188](#)

[foo]: /url

## 4.8 Paragraphs

A sequence of non-blank lines that cannot be interpreted as other kinds of blocks forms a [paragraph](#). The contents of the paragraph are the result of parsing the paragraph's raw content as inlines. The paragraph's raw content is formed by concatenating the lines and removing initial and final [whitespace](#).

A simple example with two paragraphs:

#### [Example 189](#)

aaa	<p>aaa</p>
bbb	<p>bbb</p>

Paragraphs can contain multiple lines, but no blank lines:

#### [Example 190](#)

aaa	<p>aaa
bbb	bbb</p>
ccc	<p>ccc
ddd	ddd</p>

Multiple blank lines between paragraph have no effect:

#### [Example 191](#)

aaa	<p>aaa</p>
	<p>bbb</p>
bbb	

Leading spaces are skipped:

#### [Example 192](#)

• aaa	<p>aaa
bbb	bbb</p>

Lines after the first may be indented any amount, since indented code blocks cannot interrupt paragraphs.

#### Example 193

aaa	<p>aaa
.....bbb	bbb
.....ccc	ccc</p>

However, the first line may be indented at most three spaces, or an indented code block will be triggered:

#### Example 194

..aaa	<p>aaa
bbb	bbb</p>

#### Example 195

...aaa	<pre><code>aaa
bbb	</code></pre>
	<p>bbb</p>

Final spaces are stripped before inline parsing, so a paragraph that ends with two or more spaces will not end with a [hard line break](#):

#### Example 196

aaa .....	<p>aaa 
bbb .....	bbb</p>

## 4.9 Blank lines

[Blank lines](#) between block-level elements are ignored, except for the role they play in determining whether a [list](#) is [tight](#) or [loose](#).

Blank lines at the beginning and end of the document are also ignored.

#### Example 197

..	<p>aaa</p>
	<h1>aaa</h1>
aaa	
..	
# aaa	
..	

---

## 5 Container blocks

A [container block](#) is a block that has other blocks as its contents. There are two basic kinds of container blocks: [block quotes](#) and [list items](#). [Lists](#) are meta-containers for [list items](#).

We define the syntax for container blocks recursively. The general form of the definition is:

If X is a sequence of blocks, then the result of transforming X in such-and-such a way is a container of type Y with these blocks as its content.



So, we explain what counts as a block quote or list item by explaining how these can be *generated* from their contents. This should suffice to define the syntax, although it does not give a recipe for *parsing* these constructions. (A recipe is provided below in the section entitled [A parsing strategy](#).)

## 5.1 Block quotes

A [block quote marker](#) consists of 0-3 spaces of initial indent, plus (a) the character > together with a following space, or (b) a single character > not followed by a space.

The following rules define [block quotes](#):

1. **Basic case.** If a string of lines *Ls* constitute a sequence of blocks *Bs*, then the result of prepending a [block quote marker](#) to the beginning of each line in *Ls* is a [block quote](#) containing *Bs*.
2. **Laziness.** If a string of lines *Ls* constitute a [block quote](#) with contents *Bs*, then the result of deleting the initial [block quote marker](#) from one or more lines in which the next [non-whitespace character](#) after the [block quote marker](#) is [paragraph continuation text](#) is a block quote with *Bs* as its content. [Paragraph continuation text](#) is text that will be parsed as part of the content of a paragraph, but does not occur at the beginning of the paragraph.
3. **Consecutiveness.** A document cannot contain two [block quotes](#) in a row unless there is a [blank line](#) between them.

Nothing else counts as a [block quote](#).

Here is a simple example:

### [Example 198](#)

> # Foo	<blockquote>
> bar	<h1>Foo</h1>
> baz	<p>bar
	baz</p>
	</blockquote>

The spaces after the > characters can be omitted:

### [Example 199](#)

># Foo	<blockquote>
>bar	<h1>Foo</h1>
> baz	<p>bar
	baz</p>
	</blockquote>

The > characters can be indented 1-3 spaces:

### [Example 200](#)

..> # Foo	<blockquote>
..> bar	<h1>Foo</h1>
> baz	<p>bar
	baz</p>
	</blockquote>

Four spaces gives us a code block:

#### [Example 201](#)

...> # Foo	<pre><code>&gt; # Foo
...> bar	&gt; bar
...> baz	&gt; baz
	</code></pre>

The Laziness clause allows us to omit the > before [paragraph continuation text](#):

#### [Example 202](#)

> # Foo	<blockquote>
> bar	<h1>Foo</h1>
baz	<p>bar
	baz</p>
	</blockquote>

A block quote can contain some lazy and some non-lazy continuation lines:

#### [Example 203](#)

> bar	<blockquote>
baz	<p>bar
> foo	baz
	foo</p>
	</blockquote>

Laziness only applies to lines that would have been continuations of paragraphs had they been prepended with [block quote markers](#). For example, the > cannot be omitted in the second line of

```
> foo
> ---
```

without changing the meaning:

#### [Example 204](#)

> foo	<blockquote>
---	<p>foo</p>
	</blockquote>
	<hr />

Similarly, if we omit the > in the second line of

```
> - foo
> - bar
```

then the block quote ends after the first line:

#### [Example 205](#)

```
> - foo
- bar
```

```
<blockquote>

foo

</blockquote>

bar

```

For the same reason, we can't omit the > in front of subsequent lines of an indented or fenced code block:

#### [Example 206](#)

```
>foo
...bar
```

```
<blockquote>
<pre><code>foo
</code></pre>
</blockquote>
<pre><code>bar
</code></pre>
```

#### [Example 207](#)

```
> ```
foo
```
```

```
<blockquote>
<pre><code></code></pre>
</blockquote>
<p>foo</p>
<pre><code></code></pre>
```

Note that in the following case, we have a [lazy continuation line](#):

[Example 208](#)

```
> foo
....- bar
```

```
<blockquote>
<p>foo
- bar</p>
</blockquote>
```

To see why, note that in

```
> foo
> - bar
```

the - bar is indented too far to start a list, and can't be an indented code block because indented code blocks cannot interrupt paragraphs, so it is [paragraph continuation text](#).

A block quote can be empty:

[Example 209](#)

```
>
```

```
<blockquote>
</blockquote>
```

[Example 210](#)

>	<blockquote>
> ..	</blockquote>
> .	

A block quote can have initial or final blank lines:

[Example 211](#)

>	<blockquote>
> ¶foo	<p>foo</p>
> ..	</blockquote>

A blank line always separates block quotes:

[Example 212](#)

> ¶foo	<blockquote>
	<p>foo</p>
> ¶bar	</blockquote>
	<blockquote>
	<p>bar</p>
	</blockquote>

(Most current Markdown implementations, including John Gruber's original `Markdown.pl`, will parse this example as a single block quote with two paragraphs. But it seems better to allow the author to decide whether two block quotes or one are wanted.)

Consecutiveness means that if we put these block quotes together, we get a single block quote:

[Example 213](#)

> ¶foo	<blockquote>
> ¶bar	<p>foo
	bar</p>
	</blockquote>

To get a block quote with two paragraphs, use:

[Example 214](#)

> ¶foo	<blockquote>
>	<p>foo</p>
> ¶bar	<p>bar</p>
	</blockquote>

Block quotes can interrupt paragraphs:

[Example 215](#)

foo	<p>foo</p>
> ¶bar	<blockquote>
	<p>bar</p>
	</blockquote>

In general, blank lines are not needed before or after block quotes:

[Example 216](#)

> aaa	<blockquote>
***	<p>aaa</p>
> bbb	</blockquote>
	<hr />
	<blockquote>
	<p>bbb</p>
	</blockquote>

However, because of laziness, a blank line is needed between a block quote and a following paragraph:

[Example 217](#)

> bar	<blockquote>
baz	<p>bar
	baz</p>
	</blockquote>

[Example 218](#)

> bar	<blockquote>
	<p>bar</p>
baz	</blockquote>
	<p>baz</p>

[Example 219](#)

> bar	<blockquote>
>	<p>bar</p>
baz	</blockquote>
	<p>baz</p>

It is a consequence of the Laziness rule that any number of initial >s may be omitted on a continuation line of a nested block quote:

[Example 220](#)

> > > foo	<blockquote>
bar	<blockquote>
	<blockquote>
	<p>foo
	bar</p>
	</blockquote>
	</blockquote>
	</blockquote>

[Example 221](#)

>>> foo	<blockquote>
> bar	<blockquote>
>>baz	<blockquote>
	<p>foo
	bar
	baz</p>
	</blockquote>
	</blockquote>
	</blockquote>

When including an indented code block in a block quote, remember that the [block quote marker](#) includes both the > and a following space. So *five spaces* are needed after the >:

Example 222

>code	<blockquote>
	<pre><code>code
> ...not code	</code></pre>
	</blockquote>
	<blockquote>
	<p>not code</p>
	</blockquote>

5.2 List items

A [list marker](#) is a [bullet list marker](#) or an [ordered list marker](#).

A [bullet list marker](#) is a -, +, or \* character.

An [ordered list marker](#) is a sequence of 1–9 arabic digits (0–9), followed by either a . character or a) character. (The reason for the length limit is that with 10 digits we start seeing integer overflows in some browsers.)

The following rules define [list items](#):

1. **Basic case.** If a sequence of lines *Ls* constitute a sequence of blocks *Bs* starting with a [non-whitespace character](#), and *M* is a list marker of width *W* followed by $1 \leq N \leq 4$ spaces, then the result of prepending *M* and the following spaces to the first line of *Ls*, and indenting subsequent lines of *Ls* by *W* + *N* spaces, is a list item with *Bs* as its contents. The type of the list item (bullet or ordered) is determined by the type of its list marker. If the list item is ordered, then it is also assigned a start number, based on the ordered list marker.

Exceptions:

1. When the first list item in a [list](#) interrupts a paragraph—that is, when it starts on a line that would otherwise count as [paragraph continuation text](#)—then (a) the lines *Ls* must not begin with a blank line, and (b) if the list item is ordered, the start number must be 1.
2. If any line is a [thematic break](#) then that line is not a list item.

For example, let *Ls* be the lines

Example 223

A paragraph	<p>A paragraph
with two lines.	with two lines.</p>
	<pre><code>indented code
...indented code	</code></pre>
	<blockquote>
> A block quote.	<p>A block quote.</p>
	</blockquote>

And let *M* be the marker 1. , and *N* = 2. Then rule #1 says that the following is an ordered list item with start number 1, and the same contents as *Ls*:

Example 224

1. • A paragraph	
...with two lines.	
	<p>A paragraph
	with two lines.</p>
.....indented code	<pre><code>indented code
	</code></pre>
	<blockquote>
...> A block quote.	<p>A block quote.</p>
	</blockquote>
	
	

The most important thing to notice is that the position of the text after the list marker determines how much indentation is needed in subsequent blocks in the list item. If the list marker takes up two spaces, and there are three spaces between the list marker and the next [non-whitespace character](#), then blocks must be indented five spaces in order to fall under the list item.

Here are some examples showing how far content must be indented to be put under the list item:

[Example 225](#)

- one	
	one
two	
	<p>two</p>

[Example 226](#)

- one	
	
• two	<p>one</p>
	<p>two</p>
	
	

[Example 227](#)

• • • one	
	one
• • • two	
	<pre><code> two
	</code></pre>

[Example 228](#)

• • • one	
	
• • • two	<p>one</p>
	<p>two</p>
	
	

It is tempting to think of this in terms of columns: the continuation blocks must be indented at least to the column of the first [non-whitespace character](#) after the list marker. However, that is not quite right. The spaces after the list marker determine how much relative indentation is needed. Which column this indentation reaches will depend on how the list item is embedded in other constructions, as shown by this example:

[Example 229](#)

..> 1. •one	<blockquote>
>>	<blockquote>
>>two	
	
	<p>one</p>
	<p>two</p>
	
	
	</blockquote>
	</blockquote>

Here `two` occurs in the same column as the list marker `1.`, but is actually contained in the list item, because there is sufficient indentation after the last containing blockquote marker.

The converse is also possible. In the following example, the word `two` occurs far to the right of the initial text of the list item, `one`, but it is not considered part of the list item, because it is not indented far enough past the blockquote marker:

[Example 230](#)

>>- one	<blockquote>
>>	<blockquote>
•> •> two	
	one
	
	<p>two</p>
	</blockquote>
	</blockquote>

Note that at least one space is needed between the list marker and any following content, so these are not list items:

[Example 231](#)

-one	<p>-one</p>
2. two	<p>2. two</p>

A list item may contain blocks that are separated by more than one blank line.

[Example 232](#)

- foo	
	
	<p>foo</p>
• bar	<p>bar</p>
	
	

A list item may contain any kind of block:

[Example 233](#)

1. •foo	
	
....`	<p>foo</p>
...bar	<pre><code>bar
....`	</code></pre>
	<p>baz</p>
...baz	<blockquote>
	<p>bam</p>
...> bam	</blockquote>
	
	

A list item that contains an indented code block will preserve empty lines within the code block verbatim.

[Example 234](#)

- Foo	
	
.....bar	<p>Foo</p>
	<pre><code>bar
.....baz	baz
	</code></pre>
	
	

Note that ordered list start numbers must be nine digits or less:

[Example 235](#)

123456789. ok	<ol start="123456789">
	ok
	

[Example 236](#)

1234567890. not ok	<p>1234567890. not ok</p>
--------------------	---------------------------

A start number may begin with 0s:

[Example 237](#)

0. ok	<ol start="0">
	ok
	

[Example 238](#)

003. ok	<ol start="3">
	ok
	

A start number may not be negative:

[Example 239](#)

-1. not ok

```
<p>-1. not ok</p>
```

2. **Item starting with indented code.** If a sequence of lines *Ls* constitute a sequence of blocks *Bs* starting with an indented code block, and *M* is a list marker of width *W* followed by one space, then the result of prepending *M* and the following space to the first line of *Ls*, and indenting subsequent lines of *Ls* by *W + 1* spaces, is a list item with *Bs* as its contents. If a line is empty, then it need not be indented. The type of the list item (bullet or ordered) is determined by the type of its list marker. If the list item is ordered, then it is also assigned a start number, based on the ordered list marker.

An indented code block will have to be indented four spaces beyond the edge of the region where text will be included in the list item. In the following case that is 6 spaces:

[Example 240](#)

```
- foo
    ....bar
```

```
<ul>
<li>
<p>foo</p>
<pre><code>bar
</code></pre>
</li>
</ul>
```

And in this case it is 11 spaces:

[Example 241](#)

```
•10. •foo
    .....bar
```

```
<ol start="10">
<li>
<p>foo</p>
<pre><code>bar
</code></pre>
</li>
</ol>
```

If the *first* block in the list item is an indented code block, then by rule #2, the contents must be indented *one* space after the list marker:

[Example 242](#)

```
...indented code
paragraph
...more code
```

```
<pre><code>indented code
</code></pre>
<p>paragraph</p>
<pre><code>more code
</code></pre>
```

[Example 243](#)

```
1. ....indented code
    ..paragraph
    .....more code
```

```
<ol>
<li>
<pre><code>indented code
</code></pre>
<p>paragraph</p>
<pre><code>more code
</code></pre>
</li>
</ol>
```

Note that an additional space indent is interpreted as space inside the code block:

[Example 244](#)

1.indented code	
	
..paragraph	<pre><code> indented code
	</code></pre>
.....more code	<p>paragraph</p>
	<pre><code>more code
	</code></pre>
	
	

Note that rules #1 and #2 only apply to two cases: (a) cases in which the lines to be included in a list item begin with a [non-whitespace character](#), and (b) cases in which they begin with an indented code block. In a case like the following, where the first block begins with a three-space indent, the rules do not allow us to form a list item by indenting the whole thing and prepending a list marker:

[Example 245](#)

..foo	<p>foo</p>
bar	<p>bar</p>

[Example 246](#)

- ...foo	
	foo
•bar	
	<p>bar</p>

This is not a significant restriction, because when a block begins with 1-3 spaces indent, the indentation can always be removed without a change in interpretation, allowing rule #1 to be applied. So, in the above case:

[Example 247](#)

- •foo	
	
..bar	<p>foo</p>
	<p>bar</p>
	
	

3. **Item starting with a blank line.** If a sequence of lines *Ls* starting with a single [blank line](#) constitute a (possibly empty) sequence of blocks *Bs*, not separated from each other by more than one blank line, and *M* is a list marker of width *W*, then the result of prepending *M* to the first line of *Ls*, and indenting subsequent lines of *Ls* by *W + 1* spaces, is a list item with *Bs* as its contents. If a line is empty, then it need not be indented. The type of the list item (bullet or ordered) is determined by the type of its list marker. If the list item is ordered, then it is also assigned a start number, based on the ordered list marker.

Here are some list items that start with a blank line but are not empty:

[Example 248](#)

-	
•foo	foo
-	
••`	<pre><code>bar
•bar	</code></pre>
••`	
-	
•••••baz	<pre><code>baz
	</code></pre>
	
	

When the list item starts with a blank line, the number of spaces following the list marker doesn't change the required indentation:

[Example 249](#)

- ...	
•foo	foo
	

A list item can begin with at most one blank line. In the following example, `foo` is not part of the list item:

[Example 250](#)

-	
	
•foo	
	<p>foo</p>

Here is an empty bullet list item:

[Example 251](#)

- foo	
-	foo
- bar	
	bar
	

It does not matter whether there are spaces following the [list marker](#):

[Example 252](#)

- foo	
- ...	foo
- bar	
	bar
	

Here is an empty ordered list item:

[Example 253](#)

1. foo	
2.	foo
3. bar	
	bar
	

A list may start or end with an empty list item:

[Example 254](#)

*	
	
	

However, an empty list item cannot interrupt a paragraph:

[Example 255](#)

foo	<p>foo
*	*</p>
	<p>foo
foo	1. </p>
1.	

4. **Indentation.** If a sequence of lines *Ls* constitutes a list item according to rule #1, #2, or #3, then the result of indenting each line of *Ls* by 1-3 spaces (the same for each line) also constitutes a list item with the same contents and attributes. If a line is empty, then it need not be indented.

Indented one space:

[Example 256](#)

1. •A paragraph	
....with two lines.	
	<p>A paragraph
.....indented code	with two lines.</p>
	<pre><code>indented code
....> A block quote.	</code></pre>
	<blockquote>
	<p>A block quote.</p>
	</blockquote>
	
	

Indented two spaces:

[Example 257](#)

```

1. 1. A paragraph
    with two lines.

    indented code

    A block quote.

```

```

<ol>
<li>
<p>A paragraph
with two lines.</p>
<pre><code>indented code
</code></pre>
<blockquote>
<p>A block quote.</p>
</blockquote>
</li>
</ol>

```

Indented three spaces:

[Example 258](#)

```

1. 1. A paragraph
    with two lines.

    indented code

    A block quote.

```

```

<ol>
<li>
<p>A paragraph
with two lines.</p>
<pre><code>indented code
</code></pre>
<blockquote>
<p>A block quote.</p>
</blockquote>
</li>
</ol>

```

Four spaces indent gives a code block:

[Example 259](#)

```

1. 1. A paragraph
    with two lines.

    indented code

    A block quote.

```

```

<pre><code>1. 1. A paragraph
    with two lines.

    indented code

    &gt; A block quote.
</code></pre>

```

5. **Laziness.** If a string of lines *Ls* constitute a [list item](#) with contents *Bs*, then the result of deleting some or all of the indentation from one or more lines in which the next [non-whitespace character](#) after the indentation is [paragraph continuation text](#) is a list item with the same contents and attributes. The unindented lines are called [lazy continuation lines](#).

Here is an example with [lazy continuation lines](#):

[Example 260](#)

```

1. 1. A paragraph
with two lines.

.....indented code

.....> A block quote.

```

```

<ol>
<li>
<p>A paragraph
with two lines.</p>
<pre><code>indented code
</code></pre>
<blockquote>
<p>A block quote.</p>
</blockquote>
</li>
</ol>

```

Indentation can be partially deleted:

[Example 261](#)

```

1. 1. A paragraph
...with two lines.

```

```

<ol>
<li>A paragraph
with two lines.</li>
</ol>

```

These examples show how laziness can work in nested structures:

[Example 262](#)

```

> 1. > Blockquote
continued here.

```

```

<blockquote>
<ol>
<li>
<blockquote>
<p>Blockquote
continued here.</p>
</blockquote>
</li>
</ol>
</blockquote>

```

[Example 263](#)

```

> 1. > Blockquote
> continued here.

```

```

<blockquote>
<ol>
<li>
<blockquote>
<p>Blockquote
continued here.</p>
</blockquote>
</li>
</ol>
</blockquote>

```

6. **That's all.** Nothing that is not counted as a list item by rules #1–5 counts as a [list item](#).

The rules for sublists follow from the general rules [above](#). A sublist must be indented the same number of spaces a paragraph would need to be in order to be included in the list item.

So, in this case we need two spaces indent:

[Example 264](#)

```

- foo
  •• bar
  •••• baz
  •••••• boo

```

```

<ul>
<li>foo
<ul>
<li>bar
<ul>
<li>baz
<ul>
<li>boo</li>
</ul>
</li>
</ul>
</li>
</ul>
</li>
</ul>

```

One is not enough:

[Example 265](#)

```

- foo
  • bar
  •• baz
  ••• boo

```

```

<ul>
<li>foo</li>
<li>bar</li>
<li>baz</li>
<li>boo</li>
</ul>

```

Here we need four, because the list marker is wider:

[Example 266](#)

```

10) foo
   ••• bar

```

```

<ol start="10">
<li>foo
<ul>
<li>bar</li>
</ul>
</li>
</ol>

```

Three is not enough:

[Example 267](#)

```

10) foo
   ••• bar

```

```

<ol start="10">
<li>foo</li>
</ol>
<ul>
<li>bar</li>
</ul>

```

A list may be the first block in a list item:

[Example 268](#)

- • foo

```
<ul>
<li>
<ul>
<li>foo</li>
</ul>
</li>
</ul>
```

[Example 269](#)

1. • 2. foo

```
<ol>
<li>
<ul>
<li>
<ol start="2">
<li>foo</li>
</ol>
</li>
</ul>
</li>
</ol>
```

A list item can contain a heading:

[Example 270](#)

- # Foo
- Bar
• ---
• baz

```
<ul>
<li>
<h1>Foo</h1>
</li>
<li>
<h2>Bar</h2>
baz</li>
</ul>
```

5.2.1 Motivation

John Gruber's Markdown spec says the following about list items:

1. "List markers typically start at the left margin, but may be indented by up to three spaces. List markers must be followed by one or more spaces or a tab."
2. "To make lists look nice, you can wrap items with hanging indents.... But if you don't want to, you don't have to."
3. "List items may consist of multiple paragraphs. Each subsequent paragraph in a list item must be indented by either 4 spaces or one tab."
4. "It looks nice if you indent every line of the subsequent paragraphs, but here again, Markdown will allow you to be lazy."
5. "To put a blockquote within a list item, the blockquote's > delimiters need to be indented."
6. "To put a code block within a list item, the code block needs to be indented twice — 8 spaces or two tabs."

These rules specify that a paragraph under a list item must be indented four spaces (presumably, from the left margin, rather than the start of the list marker, but this is not said), and that code under a list item must be indented eight spaces instead of the usual four. They

also say that a block quote must be indented, but not by how much; however, the example given has four spaces indentation. Although nothing is said about other kinds of block-level content, it is certainly reasonable to infer that *all* block elements under a list item, including other lists, must be indented four spaces. This principle has been called the *four-space rule*.

The four-space rule is clear and principled, and if the reference implementation `Markdown.pl` had followed it, it probably would have become the standard. However, `Markdown.pl` allowed paragraphs and sublists to start with only two spaces indentation, at least on the outer level. Worse, its behavior was inconsistent: a sublist of an outer-level list needed two spaces indentation, but a sublist of this sublist needed three spaces. It is not surprising, then, that different implementations of Markdown have developed very different rules for determining what comes under a list item. (Pandoc and `python-Markdown`, for example, stuck with Gruber's syntax description and the four-space rule, while `discount`, `redcarpet`, `marked`, `PHP Markdown`, and others followed `Markdown.pl`'s behavior more closely.)

Unfortunately, given the divergences between implementations, there is no way to give a spec for list items that will be guaranteed not to break any existing documents. However, the spec given here should correctly handle lists formatted with either the four-space rule or the more forgiving `Markdown.pl` behavior, provided they are laid out in a way that is natural for a human to read.

The strategy here is to let the width and indentation of the list marker determine the indentation necessary for blocks to fall under the list item, rather than having a fixed and arbitrary number. The writer can think of the body of the list item as a unit which gets indented to the right enough to fit the list marker (and any indentation on the list marker). (The laziness rule, #5, then allows continuation lines to be unindented if needed.)

This rule is superior, we claim, to any rule requiring a fixed level of indentation from the margin. The four-space rule is clear but unnatural. It is quite unintuitive that

```
- foo

  bar

- baz
```

should be parsed as two lists with an intervening paragraph,

```
<ul>
<li>foo</li>
</ul>
<p>bar</p>
<ul>
<li>baz</li>
</ul>
```

as the four-space rule demands, rather than a single list,

```
<ul>
<li>
<p>foo</p>
<p>bar</p>
<ul>
<li>baz</li>
</ul>
</li>
</ul>
```

The choice of four spaces is arbitrary. It can be learned, but it is not likely to be guessed, and it trips up beginners regularly.

Would it help to adopt a two-space rule? The problem is that such a rule, together with the rule allowing 1–3 spaces indentation of the initial list marker, allows text that is indented *less than* the original list marker to be included in the list item. For example, `Markdown.pl` parses

```
- one

two
```

as a single list item, with `two` a continuation paragraph:

```
<ul>
<li>
<p>one</p>
<p>two</p>
</li>
</ul>
```

and similarly

```
> - one
>
> two
```

as

```
<blockquote>
<ul>
<li>
<p>one</p>
<p>two</p>
</li>
</ul>
</blockquote>
```

This is extremely unintuitive.

Rather than requiring a fixed indent from the margin, we could require a fixed indent (say, two spaces, or even one space) from the list marker (which may itself be indented). This proposal would remove the last anomaly discussed. Unlike the spec presented above, it would count the following as a list item with a subparagraph, even though the paragraph `bar` is not indented as far as the first paragraph `foo`:

```
10. foo

    bar
```

Arguably this text does read like a list item with `bar` as a subparagraph, which may count in favor of the proposal. However, on this proposal indented code would have to be indented six spaces after the list marker. And this would break a lot of existing Markdown, which has the pattern:

```
1. foo

    indented code
```

where the code is indented eight spaces. The spec above, by contrast, will parse this text as expected, since the code block's indentation is measured from the beginning of `foo`.

The one case that needs special treatment is a list item that *starts* with indented code. How much indentation is required in that case, since we don't have a "first paragraph" to measure from? Rule #2 simply stipulates that in such cases, we require one space indentation from the list marker (and then the normal four spaces for the indented code). This will match the four-space rule in cases where the list marker plus its initial indentation takes four spaces (a common case), but diverge in other cases.

5.3 Lists

A **list** is a sequence of one or more list items [of the same type](#). The list items may be separated by any number of blank lines.

Two list items are [of the same type](#) if they begin with a [list marker](#) of the same type. Two list markers are of the same type if (a) they are bullet list markers using the same character (`-`, `+`, or `*`) or (b) they are ordered list numbers with the same delimiter (either `.` or `)`).

A list is an **ordered list** if its constituent list items begin with [ordered list markers](#), and a **bullet list** if its constituent list items begin with [bullet list markers](#).

The **start number** of an **ordered list** is determined by the list number of its initial list item. The numbers of subsequent list items are disregarded.

A list is **loose** if any of its constituent list items are separated by blank lines, or if any of its constituent list items directly contain two block-level elements with a blank line between them. Otherwise a list is **tight**. (The difference in HTML output is that paragraphs in a loose list are wrapped in `<p>` tags, while paragraphs in a tight list are not.)

Changing the bullet or ordered list delimiter starts a new list:

[Example 271](#)

<code>- foo</code>	<code></code>
<code>- bar</code>	<code>foo</code>
<code>+ baz</code>	<code>bar</code>
	<code></code>
	<code></code>
	<code>baz</code>
	<code></code>

[Example 272](#)

<code>1. foo</code>	<code></code>
<code>2. bar</code>	<code>foo</code>
<code>3) baz</code>	<code>bar</code>
	<code></code>
	<code><ol start="3"></code>
	<code>baz</code>
	<code></code>

In CommonMark, a list can interrupt a paragraph. That is, no blank line is needed to separate a paragraph from a following list:

[Example 273](#)

Foo	<p>Foo</p>
- bar	
- baz	bar
	baz
	

Markdown.pl does not allow this, through fear of triggering a list via a numeral in a hard-wrapped line:

```
The number of windows in my house is
14. The number of doors is 6.
```

Oddly, though, Markdown.pl *does* allow a blockquote to interrupt a paragraph, even though the same considerations might apply.

In CommonMark, we do allow lists to interrupt paragraphs, for two reasons. First, it is natural and not uncommon for people to start lists without blank lines:

```
I need to buy
- new shoes
- a coat
- a plane ticket
```

Second, we are attracted to a

principle of uniformity: if a chunk of text has a certain meaning, it will continue to have the same meaning when put into a container block (such as a list item or blockquote).

(Indeed, the spec for [list items](#) and [block quotes](#) presupposes this principle.) This principle implies that if

```
* I need to buy
  - new shoes
  - a coat
  - a plane ticket
```

is a list item containing a paragraph followed by a nested sublist, as all Markdown implementations agree it is (though the paragraph may be rendered without <p> tags, since the list is “tight”), then

```
I need to buy
- new shoes
- a coat
- a plane ticket
```

by itself should be a paragraph followed by a nested sublist.

Since it is well established Markdown practice to allow lists to interrupt paragraphs inside list items, the [principle of uniformity](#) requires us to allow this outside list items as well. ([reStructuredText](#) takes a different approach, requiring blank lines before lists even inside other list items.)

In order to solve of unwanted lists in paragraphs with hard-wrapped numerals, we allow only lists starting with 1 to interrupt paragraphs. Thus,

[Example 274](#)

The number of windows in my house is
14. •The number of doors is 6.

```
<p>The number of windows in my house is  
14. •The number of doors is 6.</p>
```

We may still get an unintended result in cases like

[Example 275](#)

The number of windows in my house is
1. •The number of doors is 6.

```
<p>The number of windows in my house is</p>  
<ol>  
<li>The number of doors is 6.</li>  
</ol>
```

but this rule should prevent most spurious list captures.

There can be any number of blank lines between items:

[Example 276](#)

- foo

- bar

- baz

```
<ul>  
<li>  
<p>foo</p>  
</li>  
<li>  
<p>bar</p>  
</li>  
<li>  
<p>baz</p>  
</li>  
</ul>
```

[Example 277](#)

- foo

•- bar

•••- baz

•••••bim

```
<ul>  
<li>foo  
<ul>  
<li>bar  
<ul>  
<li>  
<p>baz</p>  
<p>bim</p>  
</li>  
</ul>  
</li>  
</ul>
```

To separate consecutive lists of the same type, or to separate a list from an indented code block that would otherwise be parsed as a subparagraph of the final list item, you can insert a blank HTML comment:

[Example 278](#)

- foo	
- bar	foo
	bar
<!-- -->	
	<!-- -->
- baz	
- bim	baz
	bim
	

Example 279

- ..foo	
	
...notcode	<p>foo</p>
	<p>notcode</p>
- ..foo	
	
<!-- -->	<p>foo</p>
	
...code	
	<!-- -->
	<pre><code>code
	</code></pre>

List items need not be indented to the same level. The following list items will be treated as items at the same list level, since none is indented enough to belong to the previous list item:

Example 280

- a	
- b	a
.. c	b
... d	c
.. e	d
- f	e
- g	f
	g
	

Example 281

1. a	
	
.. 2. b	<p>a</p>
	
... 3. c	
	<p>b</p>
	
	
	<p>c</p>
	
	

Note, however, that list items may not be indented more than three spaces. Here - e is treated as a paragraph continuation line, because it is indented more than three spaces:

Example 282

- a	
- b	a
- c	b
- d	c
- e	d
	- e
	

And here, 3. c is treated as in indented code block, because it is indented four spaces and preceded by a blank line.

[Example 283](#)

1. a	
	
• 2. b	<p>a</p>
	
••• 3. c	
	<p>b</p>
	
	
	<pre><code>3. c
	</code></pre>

This is a loose list, because there is a blank line between two of the list items:

[Example 284](#)

- a	
- b	
	<p>a</p>
- c	
	
	<p>b</p>
	
	
	<p>c</p>
	
	

So is this, with a empty second item:

[Example 285](#)

* a	
*	
	<p>a</p>
* c	
	
	
	<p>c</p>
	
	

These are loose lists, even though there is no space between the items, because one of the items directly contains two block-level elements with a blank line between them:

[Example 286](#)

- a	
- b	
	<p>a</p>
• c	
- d	
	<p>b</p>
	<p>c</p>
	
	
	<p>d</p>
	
	

[Example 287](#)

- a	
- b	
	<p>a</p>
• [ref]: /url	
- d	
	<p>b</p>
	
	
	<p>d</p>
	
	

This is a tight list, because the blank lines are in a code block:

[Example 288](#)

- a	
- • ``	a
• b	
	<pre><code>b
• • ``	
- c	</code></pre>
	
	c
	

This is a tight list, because the blank line is between two paragraphs of a sublist. So the sublist is loose while the outer list is tight:

[Example 289](#)

- a	
•- b	a
	
••• c	
- d	<p>b</p>
	<p>c</p>
	
	
	
	d
	

This is a tight list, because the blank line is inside the block quote:

[Example 290](#)

* a	
•> b	a
•>	<blockquote>
* c	<p>b</p>
	</blockquote>
	
	c
	

This list is tight, because the consecutive block elements are not separated by blank lines:

[Example 291](#)

- a	
•> b	a
•>	<blockquote>
•c	<p>b</p>
•>	</blockquote>
- d	<pre><code>c
	</code></pre>
	
	d
	

A single-paragraph list is tight:

[Example 292](#)

- a	
	a
	

[Example 293](#)

- a	
•> b	a
	
	b
	
	
	

This list is loose, because of the blank line between the two block elements in the list item:

[Example 294](#)

1. •>	
•>foo	
•>	<pre><code>foo
	</code></pre>
	<p>bar</p>
	
	

Here the outer list is loose, the inner list tight:

[Example 295](#)

* foo	
•• bar	
	<p>foo</p>
	
• baz	bar
	
	<p>baz</p>
	
	

[Example 296](#)

- a	
•• b	
•• c	<p>a</p>
	
- d	b
•• e	c
•• f	
	
	
	<p>d</p>
	
	e
	f

6 Inlines

Inlines are parsed sequentially from the beginning of the character stream to the end (left to right, in left-to-right languages). Thus, for example, in

[Example 297](#)

`hi`lo`	<p><code>hi</code>lo`</p>
---------	---------------------------

hi is parsed as code, leaving the backtick at the end as a literal backtick.

6.1 Backslash escapes

Any ASCII punctuation character may be backslash-escaped:

[Example 298](#)

\\!~\"#\$%&\' \(\) \* \+ , \- \. \/ \: \; \< \= \> \?	<p>!"#%&'()*+,-./:;<=>;?
\\@\\[\\]\\^\\_\\`\\{\\ \\}\\~	@[\\]^_`{ }~</p>

Backslashes before other characters are treated as literal backslashes:

[Example 299](#)

\→\A\a\ 3\ φ \«

<p>\→\A\a\ 3\ φ \«</p>

Escaped characters are treated as regular characters and do not have their usual Markdown meanings:

Example 300

```
\*not emphasized*
\<br/> not a tag
\[not a link] (/foo)
\`not code`
1\. not a list
\* not a list
\# not a heading
\[foo]: /url "not a reference"
&ouml; not a character entity
```

```
<p>*not emphasized*
<br> not a tag
[not a link] (/foo)
`not code`
1. not a list
* not a list
# not a heading
[foo]: /url "not a reference"
&amp;ouml; not a character entity</p>
```

If a backslash is itself escaped, the following character is not:

Example 301

\\\*emphasis\*

<p>\emphasis</p>

A backslash at the end of the line is a [hard line break](#):

Example 302

```
foo\
bar
```

```
<p>foo<br />
bar</p>
```

Backslash escapes do not work in code blocks, code spans, autolinks, or raw HTML:

Example 303

```
`\`
```

```
<p><code>\`</code></p>
```

Example 304

```
...\[
```

```
<pre><code>\[
</code></pre>
```

Example 305

```
~~~
\[
~~~
```

```
<pre><code>\[
</code></pre>
```

Example 306

<http://example.com?find=\\\*

```
<p><a href="http://example.com?
find=%5C*">http://example.com?find=\\*</a>
</p>
```

Example 307

But they work in all other contexts, including URLs and link titles, link references, and [info strings](#) in [fenced code blocks](#):

Example 308

<pre>[foo](/bar\* "ti*tle")</pre>	<pre><p>foo </p></pre>
-----------------------------------	---

Example 309

<pre>[foo] [foo]: /bar\* "ti*tle"</pre>	<pre><p>foo </p></pre>
---	---

Example 310

<pre>```foo+bar foo ```</pre>	<pre><pre><code class="language-foo+bar">foo </code></pre></pre>
-------------------------------	--

6.2 Entity and numeric character references

Valid HTML entity references and numeric character references can be used in place of the corresponding Unicode character, with the following exceptions:

- Entity and character references are not recognized in code blocks and code spans.
- Entity and character references cannot stand in place of special characters that define structural elements in CommonMark. For example, although `*` can be used in place of a literal `*` character, `*` cannot replace `*` in emphasis delimiters, bullet list markers, or thematic breaks.

Conforming CommonMark parsers need not store information about whether a particular character was represented in the source using a Unicode character or an entity reference.

[Entity references](#) consist of `&` + any of the valid HTML5 entity names + `;`. The document <https://html.spec.whatwg.org/multipage/entities.json> is used as an authoritative source for the valid entity references and their corresponding code points.

Example 311

<pre>&nbsp; & &copy; &AElig; &Dcaron; &frac34; &HilbertSpace; &DifferentialD; &ClockwiseContourIntegral; &ngE;</pre>	<pre><p> & &copy; &AElig; &Dcaron; ¼ ℋ ∫ ∮ ∄</p></pre>
--	--

[Decimal numeric character references](#) consist of `&#` + a string of 1–7 arabic digits + `;`. A numeric character reference is parsed as the corresponding Unicode character. Invalid Unicode code points will be replaced by the REPLACEMENT CHARACTER (U+FFFD). For security reasons, the code point U+0000 will also be replaced by U+FFFD.

Example 312

<pre>&#35; &#1234; &#992; &#0;</pre>	<pre><p># 𐀀 𐀀 𐀀</p></pre>
--	---------------------------------------

[Hexadecimal numeric character references](#) consist of `&#` + either `X` or `x` + a string of 1–6 hexadecimal digits + `;`. They too are parsed as the corresponding Unicode character (this time

specified with a hexadecimal numeral instead of decimal).

Example 313

<code>&#X22; &#XD06; &#xcab;</code>	<code><p>&quot; 𐄂𐄃𐄄𐄅𐄆𐄇𐄈𐄉𐄊𐄋𐄌𐄍𐄎𐄏𐄐𐄑𐄒𐄓𐄔𐄕𐄖𐄗𐄘𐄙𐄚𐄛𐄜𐄝𐄞𐄟𐄠𐄡𐄢𐄣𐄤𐄥𐄦𐄧𐄨𐄩𐄪𐄫𐄬𐄭𐄮𐄯𐄰𐄱𐄲𐄳𐄴𐄵𐄶𐄷𐄸𐄹𐄺𐄻𐄼𐄽𐄾𐄿𐅀𐅁𐅂𐅃𐅄𐅅𐅆𐅇𐅈𐅉𐅊𐅋𐅌𐅍𐅎𐅏𐅐𐅑𐅒𐅓𐅔𐅕𐅖𐅗𐅘𐅙𐅚𐅛𐅜𐅝𐅞𐅟𐅠𐅡𐅢𐅣𐅤𐅥𐅦𐅧𐅨𐅩𐅪𐅫𐅬𐅭𐅮𐅯𐅰𐅱𐅲𐅳𐅴𐅵𐅶𐅷𐅸𐅹𐅺𐅻𐅼𐅽𐅾𐅿𐆀𐆁𐆂𐆃𐆄𐆅𐆆𐆇𐆈𐆉𐆊𐆋𐆌𐆍𐆎𐆏𐆐𐆑𐆒𐆓𐆔𐆕𐆖𐆗𐆘𐆙𐆚𐆛𐆜𐆝𐆞𐆟𐆠𐆡𐆢𐆣𐆤𐆥𐆦𐆧𐆨𐆩𐆪𐆫𐆬𐆭𐆮𐆯𐆰𐆱𐆲𐆳𐆴𐆵𐆶𐆷𐆸𐆹𐆺𐆻𐆼𐆽𐆾𐆿𐇀𐇁𐇂𐇃𐇄𐇅𐇆𐇇𐇈𐇉𐇊𐇋𐇌𐇍𐇎𐇏𐇐𐇑𐇒𐇓𐇔𐇕𐇖𐇗𐇘𐇙𐇚𐇛𐇜𐇝𐇞𐇟𐇠𐇡𐇢𐇣𐇤𐇥𐇦𐇧𐇨𐇩𐇪𐇫𐇬𐇭𐇮𐇯𐇰𐇱𐇲𐇳𐇴𐇵𐇶𐇷𐇸𐇹𐇺𐇻𐇼𐇽𐇾𐇿𐈀𐈁𐈂𐈃𐈄𐈅𐈆𐈇𐈈𐈉𐈊𐈋𐈌𐈍𐈎𐈏𐈐𐈑𐈒𐈓𐈔𐈕𐈖𐈗𐈘𐈙𐈚𐈛𐈜𐈝𐈞𐈟𐈠𐈡𐈢𐈣𐈤𐈥𐈦𐈧𐈨𐈩𐈪𐈫𐈬𐈭𐈮𐈯𐈰𐈱𐈲𐈳𐈴𐈵𐈶𐈷𐈸𐈹𐈺𐈻𐈼𐈽𐈾𐈿𐉀𐉁𐉂𐉃𐉄𐉅𐉆𐉇𐉈𐉉𐉊𐉋𐉌𐉍𐉎𐉏𐉐𐉑𐉒𐉓𐉔𐉕𐉖𐉗𐉘𐉙𐉚𐉛𐉜𐉝𐉞𐉟𐉠𐉡𐉢𐉣𐉤𐉥𐉦𐉧𐉨𐉩𐉪𐉫𐉬𐉭𐉮𐉯𐉰𐉱𐉲𐉳𐉴𐉵𐉶𐉷𐉸𐉹𐉺𐉻𐉼𐉽𐉾𐉿𐊀𐊁𐊂𐊃𐊄𐊅𐊆𐊇𐊈𐊉𐊊𐊋𐊌𐊍𐊎𐊏𐊐𐊑𐊒𐊓𐊔𐊕𐊖𐊗𐊘𐊙𐊚𐊛𐊜𐊝𐊞𐊟𐊠𐊡𐊢𐊣𐊤𐊥𐊦𐊧𐊨𐊩𐊪𐊫𐊬𐊭𐊮𐊯𐊰𐊱𐊲𐊳𐊴𐊵𐊶𐊷𐊸𐊹𐊺𐊻𐊼𐊽𐊾𐊿𐋀𐋁𐋂𐋃𐋄𐋅𐋆𐋇𐋈𐋉𐋊𐋋𐋌𐋍𐋎𐋏𐋐𐋑𐋒𐋓𐋔𐋕𐋖𐋗𐋘𐋙𐋚𐋛𐋜𐋝𐋞𐋟𐋠𐋡𐋢𐋣𐋤𐋥𐋦𐋧𐋨𐋩𐋪𐋫𐋬𐋭𐋮𐋯𐋰𐋱𐋲𐋳𐋴𐋵𐋶𐋷𐋸𐋹𐋺𐋻𐋼𐋽𐋾𐋿𐌀𐌁𐌂𐌃𐌄𐌅𐌆𐌇𐌈𐌉𐌊𐌋𐌌𐌍𐌎𐌏𐌐𐌑𐌒𐌓𐌔𐌕𐌖𐌗𐌘𐌙𐌚𐌛𐌜𐌝𐌞𐌟𐌠𐌡𐌢𐌣𐌤𐌥𐌦𐌧𐌨𐌩𐌪𐌫𐌬𐌭𐌮𐌯𐌰𐌱𐌲𐌳𐌴𐌵𐌶𐌷𐌸𐌹𐌺𐌻𐌼𐌽𐌾𐌿𐍀𐍁𐍂𐍃𐍄𐍅𐍆𐍇𐍈𐍉𐍊𐍋𐍌𐍍𐍎𐍏𐍐𐍑𐍒𐍓𐍔𐍕𐍖𐍗𐍘𐍙𐍚𐍛𐍜𐍝𐍞𐍟𐍠𐍡𐍢𐍣𐍤𐍥𐍦𐍧𐍨𐍩𐍪𐍫𐍬𐍭𐍮𐍯𐍰𐍱𐍲𐍳𐍴𐍵𐍶𐍷𐍸𐍹𐍺𐍻𐍼𐍽𐍾𐍿𐎀𐎁𐎂𐎃𐎄𐎅𐎆𐎇𐎈𐎉𐎊𐎋𐎌𐎍𐎎𐎏𐎐𐎑𐎒𐎓𐎔𐎕𐎖𐎗𐎘𐎙𐎚𐎛𐎜𐎝𐎞𐎟𐎠𐎡𐎢𐎣𐎤𐎥𐎦𐎧𐎨𐎩𐎪𐎫𐎬𐎭𐎮𐎯𐎰𐎱𐎲𐎳𐎴𐎵𐎶𐎷𐎸𐎹𐎺𐎻𐎼𐎽𐎾𐎿𐏀𐏁𐏂𐏃𐏄𐏅𐏆𐏇𐏈𐏉𐏊𐏋𐏌𐏍𐏎𐏏𐏐𐏑𐏒𐏓𐏔𐏕𐏖𐏗𐏘𐏙𐏚𐏛𐏜𐏝𐏞𐏟𐏠𐏡𐏢𐏣𐏤𐏥𐏦𐏧𐏨𐏩𐏪𐏫𐏬𐏭𐏮𐏯𐏰𐏱𐏲𐏳𐏴𐏵𐏶𐏷𐏸𐏹𐏺𐏻𐏼𐏽𐏾𐏿𐐀𐐁𐐂𐐃𐐄𐐅𐐆𐐇𐐈𐐉𐐊𐐋𐐌𐐍𐐎𐐏𐐐𐐑𐐒𐐓𐐔𐐕𐐖𐐗𐐘𐐙𐐚𐐛𐐜𐐝𐐞𐐟𐐠𐐡𐐢𐐣𐐤𐐥𐐦𐐧𐐨𐐩𐐪𐐫𐐬𐐭𐐮𐐯𐐰𐐱𐐲𐐳𐐴𐐵𐐶𐐷𐐸𐐹𐐺𐐻𐐼𐐽𐐾𐐿𐑀𐑁𐑂𐑃𐑄𐑅𐑆𐑇𐑈𐑉𐑊𐑋𐑌𐑍𐑎𐑏𐑐𐑑𐑒𐑓𐑔𐑕𐑖𐑗𐑘𐑙𐑚𐑛𐑜𐑝𐑞𐑟𐑠𐑡𐑢𐑣𐑤𐑥𐑦𐑧𐑨𐑩𐑪𐑫𐑬𐑭𐑮𐑯𐑰𐑱𐑲𐑳𐑴𐑵𐑶𐑷𐑸𐑹𐑺𐑻𐑼𐑽𐑾𐑿𐒀𐒁𐒂𐒃𐒄𐒅𐒆𐒇𐒈𐒉𐒊𐒋𐒌𐒍𐒎𐒏𐒐𐒑𐒒𐒓𐒔𐒕𐒖𐒗𐒘𐒙𐒚𐒛𐒜𐒝𐒞𐒟𐒠𐒡𐒢𐒣𐒤𐒥𐒦𐒧𐒨𐒩𐒪𐒫𐒬𐒭𐒮𐒯𐒰𐒱𐒲𐒳𐒴𐒵𐒶𐒷𐒸𐒹𐒺𐒻𐒼𐒽𐒾𐒿𐓀𐓁𐓂𐓃𐓄𐓅𐓆𐓇𐓈𐓉𐓊𐓋𐓌𐓍𐓎𐓏𐓐𐓑𐓒𐓓𐓔𐓕𐓖𐓗𐓘𐓙𐓚𐓛𐓜𐓝𐓞𐓟𐓠𐓡𐓢𐓣𐓤𐓥𐓦𐓧𐓨𐓩𐓪𐓫𐓬𐓭𐓮𐓯𐓰𐓱𐓲𐓳𐓴𐓵𐓶𐓷𐓸𐓹𐓺𐓻𐓼𐓽𐓾𐓿𐔀𐔁𐔂𐔃𐔄𐔅𐔆𐔇𐔈𐔉𐔊𐔋𐔌𐔍𐔎𐔏𐔐𐔑𐔒𐔓𐔔𐔕𐔖𐔗𐔘𐔙𐔚𐔛𐔜𐔝𐔞𐔟𐔠𐔡𐔢𐔣𐔤𐔥𐔦𐔧𐔨𐔩𐔪𐔫𐔬𐔭𐔮𐔯𐔰𐔱𐔲𐔳𐔴𐔵𐔶𐔷𐔸𐔹𐔺𐔻𐔼𐔽𐔾𐔿𐕀𐕁𐕂𐕃𐕄𐕅𐕆𐕇𐕈𐕉𐕊𐕋𐕌𐕍𐕎𐕏𐕐𐕑𐕒𐕓𐕔𐕕𐕖𐕗𐕘𐕙𐕚𐕛𐕜𐕝𐕞𐕟𐕠𐕡𐕢𐕣𐕤𐕥𐕦𐕧𐕨𐕩𐕪𐕫𐕬𐕭𐕮𐕯𐕰𐕱𐕲𐕳𐕴𐕵𐕶𐕷𐕸𐕹𐕺𐕻𐕼𐕽𐕾𐕿𐖀𐖁𐖂𐖃𐖄𐖅𐖆𐖇𐖈𐖉𐖊𐖋𐖌𐖍𐖎𐖏𐖐𐖑𐖒𐖓𐖔𐖕𐖖𐖗𐖘𐖙𐖚𐖛𐖜𐖝𐖞𐖟𐖠𐖡𐖢𐖣𐖤𐖥𐖦𐖧𐖨𐖩𐖪𐖫𐖬𐖭𐖮𐖯𐖰𐖱𐖲𐖳𐖴𐖵𐖶𐖷𐖸𐖹𐖺𐖻𐖼𐖽𐖾𐖿𐗀𐗁𐗂𐗃𐗄𐗅𐗆𐗇𐗈𐗉𐗊𐗋𐗌𐗍𐗎𐗏𐗐𐗑𐗒𐗓𐗔𐗕𐗖𐗗𐗘𐗙𐗚𐗛𐗜𐗝𐗞𐗟𐗠𐗡𐗢𐗣𐗤𐗥𐗦𐗧𐗨𐗩𐗪𐗫𐗬𐗭𐗮𐗯𐗰𐗱𐗲𐗳𐗴𐗵𐗶𐗷𐗸𐗹𐗺𐗻𐗼𐗽𐗾𐗿𐘀𐘁𐘂𐘃𐘄𐘅𐘆𐘇𐘈𐘉𐘊𐘋𐘌𐘍𐘎𐘏𐘐𐘑𐘒𐘓𐘔𐘕𐘖𐘗𐘘𐘙𐘚𐘛𐘜𐘝𐘞𐘟𐘠𐘡𐘢𐘣𐘤𐘥𐘦𐘧𐘨𐘩𐘪𐘫𐘬𐘭𐘮𐘯𐘰𐘱𐘲𐘳𐘴𐘵𐘶𐘷𐘸𐘹𐘺𐘻𐘼𐘽𐘾𐘿𐙀𐙁𐙂𐙃𐙄𐙅𐙆𐙇𐙈𐙉𐙊𐙋𐙌𐙍𐙎𐙏𐙐𐙑𐙒𐙓𐙔𐙕𐙖𐙗𐙘𐙙𐙚𐙛𐙜𐙝𐙞𐙟𐙠𐙡𐙢𐙣𐙤𐙥𐙦𐙧𐙨𐙩𐙪𐙫𐙬𐙭𐙮𐙯𐙰𐙱𐙲𐙳𐙴𐙵𐙶𐙷𐙸𐙹𐙺𐙻𐙼𐙽𐙾𐙿𐚀𐚁𐚂𐚃𐚄𐚅𐚆𐚇𐚈𐚉𐚊𐚋𐚌𐚍𐚎𐚏𐚐𐚑𐚒𐚓𐚔𐚕𐚖𐚗𐚘𐚙𐚚𐚛𐚜𐚝𐚞𐚟𐚠𐚡𐚢𐚣𐚤𐚥𐚦𐚧𐚨𐚩𐚪𐚫𐚬𐚭𐚮𐚯𐚰𐚱𐚲𐚳𐚴𐚵𐚶𐚷𐚸𐚹𐚺𐚻𐚼𐚽𐚾𐚿𐛀𐛁𐛂𐛃𐛄𐛅𐛆𐛇𐛈𐛉𐛊𐛋𐛌𐛍𐛎𐛏𐛐𐛑𐛒𐛓𐛔𐛕𐛖𐛗𐛘𐛙𐛚𐛛𐛜𐛝𐛞𐛟𐛠𐛡𐛢𐛣𐛤𐛥𐛦𐛧𐛨𐛩𐛪𐛫𐛬𐛭𐛮𐛯𐛰𐛱𐛲𐛳𐛴𐛵𐛶𐛷𐛸𐛹𐛺𐛻𐛼𐛽𐛾𐛿𐜀𐜁𐜂𐜃𐜄𐜅𐜆𐜇𐜈𐜉𐜊𐜋𐜌𐜍𐜎𐜏𐜐𐜑𐜒𐜓𐜔𐜕𐜖𐜗𐜘𐜙𐜚𐜛𐜜𐜝𐜞𐜟𐜠𐜡𐜢𐜣𐜤𐜥𐜦𐜧𐜨𐜩𐜪𐜫𐜬𐜭𐜮𐜯𐜰𐜱𐜲𐜳𐜴𐜵𐜶𐜷𐜸𐜹𐜺𐜻𐜼𐜽𐜾𐜿𐝀𐝁𐝂𐝃𐝄𐝅𐝆𐝇𐝈𐝉𐝊𐝋𐝌𐝍𐝎𐝏𐝐𐝑𐝒𐝓𐝔𐝕𐝖𐝗𐝘𐝙𐝚𐝛𐝜𐝝𐝞𐝟𐝠𐝡𐝢𐝣𐝤𐝥𐝦𐝧𐝨𐝩𐝪𐝫𐝬𐝭𐝮𐝯𐝰𐝱𐝲𐝳𐝴𐝵𐝶𐝷𐝸𐝹𐝺𐝻𐝼𐝽𐝾𐝿𐞀𐞁𐞂𐞃𐞄𐞅𐞆𐞇𐞈𐞉𐞊𐞋𐞌𐞍𐞎𐞏𐞐𐞑𐞒𐞓𐞔𐞕𐞖𐞗𐞘𐞙𐞚𐞛𐞜𐞝𐞞𐞟𐞠𐞡𐞢𐞣𐞤𐞥𐞦𐞧𐞨𐞩𐞪𐞫𐞬𐞭𐞮𐞯𐞰𐞱𐞲𐞳𐞴𐞵𐞶𐞷𐞸𐞹𐞺𐞻𐞼𐞽𐞾𐞿𐟀𐟁𐟂𐟃𐟄𐟅𐟆𐟇𐟈𐟉𐟊𐟋𐟌𐟍𐟎𐟏𐟐𐟑𐟒𐟓𐟔𐟕𐟖𐟗𐟘𐟙𐟚𐟛𐟜𐟝𐟞𐟟𐟠𐟡𐟢𐟣𐟤𐟥𐟦𐟧𐟨𐟩𐟪𐟫𐟬𐟭𐟮𐟯𐟰𐟱𐟲𐟳𐟴𐟵𐟶𐟷𐟸𐟹𐟺𐟻𐟼𐟽𐟾𐟿𐠀𐠁𐠂𐠃𐠄𐠅𐠆𐠇𐠈𐠉𐠊𐠋𐠌𐠍𐠎𐠏𐠐𐠑𐠒𐠓𐠔𐠕𐠖𐠗𐠘𐠙𐠚𐠛𐠜𐠝𐠞𐠟𐠠𐠡𐠢𐠣𐠤𐠥𐠦𐠧𐠨𐠩𐠪𐠫𐠬𐠭𐠮𐠯𐠰𐠱𐠲𐠳𐠴𐠵𐠶𐠷𐠸𐠹𐠺𐠻𐠼𐠽𐠾𐠿𐡀𐡁𐡂𐡃𐡄𐡅𐡆𐡇𐡈𐡉𐡊𐡋𐡌𐡍𐡎𐡏𐡐𐡑𐡒𐡓𐡔𐡕𐡖𐡗𐡘𐡙𐡚𐡛𐡜𐡝𐡞𐡟𐡠𐡡𐡢𐡣𐡤𐡥𐡦𐡧𐡨𐡩𐡪𐡫𐡬𐡭𐡮𐡯𐡰𐡱𐡲𐡳𐡴𐡵𐡶𐡷𐡸𐡹𐡺𐡻𐡼𐡽𐡾𐡿𐢀𐢁𐢂𐢃𐢄𐢅𐢆𐢇𐢈𐢉𐢊𐢋𐢌𐢍𐢎𐢏𐢐𐢑𐢒𐢓𐢔𐢕𐢖𐢗𐢘𐢙𐢚𐢛𐢜𐢝𐢞𐢟𐢠𐢡𐢢𐢣𐢤𐢥𐢦𐢧𐢨𐢩𐢪𐢫𐢬𐢭𐢮𐢯𐢰𐢱𐢲𐢳𐢴𐢵𐢶𐢷𐢸𐢹𐢺𐢻𐢼𐢽𐢾𐢿𐣀𐣁𐣂𐣃𐣄𐣅𐣆𐣇𐣈𐣉𐣊𐣋𐣌𐣍𐣎𐣏𐣐𐣑𐣒𐣓𐣔𐣕𐣖𐣗𐣘𐣙𐣚𐣛𐣜𐣝𐣞𐣟𐣠𐣡𐣢𐣣𐣤𐣥𐣦𐣧𐣨𐣩𐣪𐣫𐣬𐣭𐣮𐣯𐣰𐣱𐣲𐣳𐣴𐣵𐣶𐣷𐣸𐣹𐣺𐣻𐣼𐣽𐣾𐣿𐤀𐤁𐤂𐤃𐤄𐤅𐤆𐤇𐤈𐤉𐤊𐤋𐤌𐤍𐤎𐤏𐤐𐤑𐤒𐤓𐤔𐤕𐤖𐤗𐤘𐤙𐤚𐤛𐤜𐤝𐤞𐤟𐤠𐤡𐤢𐤣𐤤𐤥𐤦𐤧𐤨𐤩𐤪𐤫𐤬𐤭𐤮𐤯𐤰𐤱𐤲𐤳𐤴𐤵𐤶𐤷𐤸𐤹𐤺𐤻𐤼𐤽𐤾𐤿𐥀𐥁𐥂𐥃𐥄𐥅𐥆𐥇𐥈𐥉𐥊𐥋𐥌𐥍𐥎𐥏𐥐𐥑𐥒𐥓𐥔𐥕𐥖𐥗𐥘𐥙𐥚𐥛𐥜𐥝𐥞𐥟𐥠𐥡𐥢𐥣𐥤𐥥𐥦𐥧𐥨𐥩𐥪𐥫𐥬𐥭𐥮𐥯𐥰𐥱𐥲𐥳𐥴𐥵𐥶𐥷𐥸𐥹𐥺𐥻𐥼𐥽𐥾𐥿𐦀𐦁𐦂𐦃𐦄𐦅𐦆𐦇𐦈𐦉𐦊𐦋𐦌𐦍𐦎𐦏𐦐𐦑𐦒𐦓𐦔𐦕𐦖𐦗𐦘𐦙𐦚𐦛𐦜𐦝𐦞𐦟𐦠𐦡𐦢𐦣𐦤𐦥𐦦𐦧𐦨𐦩𐦪𐦫𐦬𐦭𐦮𐦯𐦰𐦱𐦲𐦳𐦴𐦵𐦶𐦷𐦸𐦹𐦺𐦻𐦼𐦽𐦾𐦿𐧀𐧁𐧂𐧃𐧄𐧅𐧆𐧇𐧈𐧉𐧊𐧋𐧌𐧍𐧎𐧏𐧐𐧑𐧒𐧓𐧔𐧕𐧖𐧗𐧘𐧙𐧚𐧛𐧜𐧝𐧞𐧟𐧠𐧡𐧢𐧣𐧤𐧥𐧦𐧧𐧨𐧩𐧪𐧫𐧬𐧭𐧮𐧯𐧰𐧱𐧲𐧳𐧴𐧵𐧶𐧷𐧸𐧹𐧺𐧻𐧼𐧽𐧾𐧿𐨀𐨁𐨂𐨃𐨄𐨅𐨆𐨇𐨈𐨉𐨊𐨋𐨌𐨍𐨎𐨏𐨐𐨑𐨒𐨓𐨔𐨕𐨖𐨗𐨘𐨙𐨚𐨛𐨜𐨝𐨞𐨟𐨠𐨡𐨢𐨣𐨤𐨥𐨦𐨧𐨨𐨩𐨪𐨫𐨬𐨭𐨮𐨯𐨰𐨱𐨲𐨳𐨴𐨵𐨶𐨷𐨹𐨺𐨸𐨻𐨼𐨽𐨾𐨿𐩀𐩁𐩂𐩃𐩄𐩅𐩆𐩇𐩈𐩉𐩊𐩋𐩌𐩍𐩎𐩏𐩐𐩑𐩒𐩓𐩔𐩕𐩖𐩗𐩘𐩙𐩚𐩛𐩜𐩝𐩞𐩟𐩠𐩡𐩢𐩣𐩤𐩥𐩦𐩧𐩨𐩩𐩪𐩫𐩬𐩭𐩮𐩯𐩰𐩱𐩲𐩳𐩴𐩵𐩶𐩷𐩸𐩹𐩺𐩻𐩼𐩽𐩾𐩿𐪀𐪁𐪂𐪃𐪄𐪅𐪆𐪇𐪈𐪉𐪊𐪋𐪌𐪍𐪎𐪏𐪐𐪑𐪒𐪓𐪔𐪕𐪖𐪗𐪘𐪙𐪚𐪛𐪜𐪝𐪞𐪟𐪠𐪡𐪢𐪣𐪤𐪥𐪦𐪧𐪨𐪩𐪪𐪫𐪬𐪭𐪮𐪯𐪰𐪱𐪲𐪳𐪴𐪵𐪶𐪷𐪸𐪹𐪺𐪻𐪼𐪽𐪾𐪿𐫀𐫁𐫂𐫃𐫄𐫅𐫆𐫇𐫈𐫉𐫊𐫋𐫌𐫍𐫎𐫏𐫐𐫑𐫒𐫓𐫔𐫕𐫖𐫗𐫘𐫙𐫚𐫛𐫜𐫝𐫞𐫟𐫠𐫡𐫢𐫣𐫤𐫦𐫥𐫧𐫨𐫩𐫪𐫫𐫬𐫭𐫮𐫯𐫰𐫱𐫲𐫳𐫴𐫵𐫶𐫷𐫸𐫹𐫺𐫻𐫼𐫽𐫾𐫿𐬀𐬁𐬂𐬃𐬄𐬅𐬆𐬇𐬈𐬉𐬊𐬋𐬌𐬍𐬎𐬏𐬐𐬑𐬒𐬓𐬔𐬕𐬖𐬗𐬘𐬙𐬚𐬛𐬜𐬝𐬞𐬟𐬠𐬡𐬢𐬣𐬤𐬥𐬦𐬧𐬨𐬩𐬪𐬫𐬬𐬭𐬮𐬯𐬰𐬱𐬲𐬳𐬴𐬵𐬶𐬷𐬸𐬹𐬺𐬻𐬼𐬽𐬾𐬿𐭀𐭁𐭂𐭃𐭄𐭅𐭆𐭇𐭈𐭉𐭊𐭋𐭌𐭍𐭎𐭏𐭐𐭑𐭒𐭓𐭔𐭕𐭖𐭗𐭘𐭙𐭚𐭛𐭜𐭝𐭞𐭟𐭠𐭡𐭢𐭣𐭤𐭥𐭦𐭧𐭨𐭩𐭪𐭫𐭬𐭭𐭮𐭯𐭰𐭱𐭲𐭳𐭴𐭵𐭶𐭷𐭸𐭹𐭺𐭻𐭼𐭽𐭾𐭿𐮀𐮁𐮂𐮃𐮄𐮅𐮆𐮇𐮈𐮉𐮊𐮋𐮌𐮍𐮎𐮏𐮐𐮑𐮒𐮓𐮔𐮕𐮖𐮗𐮘𐮙𐮚𐮛𐮜𐮝𐮞𐮟𐮠𐮡𐮢𐮣𐮤𐮥𐮦𐮧𐮨𐮩𐮪𐮫𐮬𐮭𐮮𐮯𐮰𐮱𐮲𐮳𐮴𐮵𐮶𐮷𐮸𐮹𐮺𐮻𐮼𐮽𐮾𐮿𐯀𐯁𐯂𐯃𐯄𐯅𐯆𐯇𐯈𐯉𐯊𐯋𐯌𐯍𐯎𐯏𐯐𐯑𐯒𐯓𐯔𐯕𐯖𐯗𐯘𐯙𐯚𐯛𐯜𐯝𐯞𐯟𐯠𐯡𐯢𐯣𐯤𐯥𐯦𐯧𐯨𐯩𐯪𐯫𐯬𐯭𐯮𐯯𐯰𐯱𐯲𐯳𐯴𐯵𐯶𐯷𐯸𐯹𐯺𐯻𐯼𐯽𐯾𐯿𐰀𐰁𐰂𐰃𐰄𐰅𐰆𐰇𐰈𐰉𐰊𐰋𐰌𐰍𐰎𐰏𐰐𐰑𐰒𐰓𐰔𐰕𐰖𐰗𐰘𐰙𐰚𐰛𐰜𐰝𐰞𐰟𐰠𐰡𐰢𐰣𐰤𐰥𐰦𐰧𐰨𐰩𐰪𐰫𐰬𐰭𐰮𐰯𐰰𐰱𐰲𐰳𐰴𐰵𐰶𐰷𐰸𐰹𐰺𐰻𐰼𐰽𐰾𐰿𐱀𐱁𐱂𐱃𐱄𐱅𐱆𐱇𐱈𐱉𐱊𐱋𐱌𐱍𐱎𐱏𐱐𐱑𐱒𐱓𐱔𐱕𐱖𐱗𐱘𐱙𐱚𐱛𐱜𐱝𐱞𐱟𐱠𐱡𐱢𐱣𐱤𐱥𐱦𐱧𐱨𐱩𐱪𐱫𐱬𐱭𐱮𐱯𐱰𐱱𐱲𐱳𐱴𐱵𐱶𐱷𐱸𐱹𐱺𐱻𐱼𐱽𐱾𐱿𐲀𐲁𐲂𐲃𐲄𐲅𐲆𐲇𐲈𐲉𐲊𐲋𐲌𐲍𐲎𐲏𐲐𐲑𐲒𐲓𐲔𐲕𐲖𐲗𐲘𐲙𐲚𐲛𐲜𐲝𐲞𐲟𐲠𐲡𐲢𐲣𐲤𐲥𐲦𐲧𐲨𐲩𐲪𐲫𐲬𐲭𐲮𐲯𐲰𐲱𐲲𐲳𐲴𐲵𐲶𐲷𐲸𐲹𐲺𐲻𐲼𐲽𐲾𐲿𐳀𐳁𐳂𐳃𐳄𐳅𐳆𐳇𐳈𐳉𐳊𐳋𐳌𐳍𐳎𐳏𐳐𐳑𐳒𐳓𐳔𐳕𐳖𐳗𐳘𐳙𐳚𐳛𐳜𐳝𐳞𐳟𐳠𐳡𐳢𐳣𐳤𐳥𐳦𐳧𐳨𐳩𐳪𐳫𐳬𐳭𐳮𐳯𐳰𐳱</code>
---	---

Example 321

`fölöl;`

`<p><code>f&ouml;l&ouml;l;</code></p>`

Example 322

`...föl;föl;`

`<pre><code>f&ouml;l;f&ouml;l;</code></pre>`

Entity and numeric character references cannot be used in place of symbols indicating structure in CommonMark documents.

Example 323

`*foo*
foo`

`<p>*foo*
foo</p>`

Example 324

`* foo`

`<p>* foo</p>`

`* foo`

`
foo
`

Example 325

`foo

bar`

`<p>foo`

`bar</p>`

Example 326

`	foo`

`<p>→foo</p>`

Example 327

`[a] (url "tit")`

`<p>[a] (url "tit")</p>`

6.3 Code spans

A **backtick string** is a string of one or more backtick characters (```) that is neither preceded nor followed by a backtick.

A **code span** begins with a backtick string and ends with a backtick string of equal length. The contents of the code span are the characters between the two backtick strings, normalized in the following ways:

- First, [line endings](#) are converted to [spaces](#).
- If the resulting string both begins *and* ends with a [space](#) character, but does not consist entirely of [space](#) characters, a single [space](#) character is removed from the front and back. This allows you to include code that begins or ends with backtick characters, which must be separated by whitespace from the opening or closing backtick strings.

This is a simple code span:

Example 328

``foo``

`<p><code>foo</code></p>`

Here two backticks are used, because the code contains a backtick. This example also illustrates stripping of a single leading and trailing space:

[Example 329](#)

```
``foo`bar`  

<p><code>foo`bar</code></p>
```

This example shows the motivation for stripping leading and trailing spaces:

[Example 330](#)

```
` ` ` `
<p><code>` `</code></p>
```

Note that only *one* space is stripped:

[Example 331](#)

```
` ` ` `
<p><code>` `</code></p>
```

The stripping only happens if the space is on both sides of the string:

[Example 332](#)

```
`a`
<p><code> a</code></p>
```

Only [spaces](#), and not [unicode whitespace](#) in general, are stripped in this way:

[Example 333](#)

```
` b `
<p><code> b </code></p>
```

No stripping occurs if the code span contains only spaces:

[Example 334](#)

```
` `
` `
<p><code> </code>
<code> •</code></p>
```

[Line endings](#) are treated like spaces:

[Example 335](#)

```
` `
foo
bar ••
baz
` `
<p><code>foo`bar`••`baz</code></p>
```

[Example 336](#)

```
` `
foo`
` `
<p><code>foo`</code></p>
```


Interior spaces are not collapsed:

[Example 337](#)

```
`foo ••bar •
baz`
```

```
<p><code>foo ••bar •baz</code></p>
```

Note that browsers will typically collapse consecutive spaces when rendering `<code>` elements, so it is recommended that the following CSS be used:

```
code{white-space: pre-wrap;}
```

Note that backslash escapes do not work in code spans. All backslashes are treated literally:

[Example 338](#)

```
`foo\`bar`
```

```
<p><code>foo\</code>bar`</p>
```

Backslash escapes are never needed, because one can always choose a string of n backtick characters as delimiters, where the code does not contain any strings of exactly n backtick characters.

[Example 339](#)

```
` `foo`bar` `
```

```
<p><code>foo`bar</code></p>
```

[Example 340](#)

```
`foo •` `bar •`
```

```
<p><code>foo •` `bar</code></p>
```

Code span backticks have higher precedence than any other inline constructs except HTML tags and autolinks. Thus, for example, this is not parsed as emphasized text, since the second `*` is part of a code span:

[Example 341](#)

```
*foo`*`
```

```
<p>*foo<code>*</code></p>
```

And this is not parsed as a link:

[Example 342](#)

```
[not a `link](/foo`)
```

```
<p>[not a <code>link](/foo</code>)</p>
```

Code spans, HTML tags, and autolinks have the same precedence. Thus, this is code:

[Example 343](#)

```
`<a href=""`>`
```

```
<p><code>&lt;a href=&quot;&quot;&gt;&lt;/code>&quot;&gt;&lt;/p>
```

But this is an HTML tag:

[Example 344](#)

```
<a href=""`>`
```

```
<p><a href=""`>`</p>
```

And this is code:

[Example 345](#)

```
`<http://foo.bar.`baz`
```

```
<p><code>&lt;http://foo.bar.
</code>baz&gt;`</p>
```

But this is an autolink:

[Example 346](#)

```
<http://foo.bar.`baz`
```

```
<p><a •
href="http://foo.bar.%60baz">http://foo.bar.
`baz</a>`</p>
```

When a backtick string is not closed by a matching backtick string, we just have literal backticks:

[Example 347](#)

```
```foo`
```

```
<p>```foo`</p>
```

#### [Example 348](#)

```
`foo
```

```
<p>`foo</p>
```

The following case also illustrates the need for opening and closing backtick strings to be equal in length:

#### [Example 349](#)

```
`foo`bar`
```

```
<p>`foo<code>bar</code></p>
```

## 6.4 Emphasis and strong emphasis

John Gruber's original [Markdown syntax description](#) says:

Markdown treats asterisks (\*) and underscores (\_) as indicators of emphasis. Text wrapped with one \* or \_ will be wrapped with an HTML `<em>` tag; double \*'s or \_'s will be wrapped with an HTML `<strong>` tag.

This is enough for most users, but these rules leave much undecided, especially when it comes to nested emphasis. The original Markdown.pl test suite makes it clear that triple \*\*\* and \_\_\_ delimiters can be used for strong emphasis, and most implementations have also allowed the following patterns:

```
strong emph
***strong** in emph*
***emph* in strong**
in strong *emph*
*in emph **strong***
```

The following patterns are less widely supported, but the intent is clear and they are useful (especially in contexts like bibliography entries):

```
*emph *with emph* in it*
strong **with strong in it**
```

Many implementations have also restricted intraword emphasis to the `*` forms, to avoid unwanted emphasis in words containing internal underscores. (It is best practice to put these in code spans, but users often do not.)

```
internal emphasis: foo*bar*baz
no emphasis: foo_bar_baz
```

The rules given below capture all of these patterns, while allowing for efficient parsing strategies that do not backtrack.

First, some definitions. A [delimiter run](#) is either a sequence of one or more `*` characters that is not preceded or followed by a non-backslash-escaped `*` character, or a sequence of one or more `_` characters that is not preceded or followed by a non-backslash-escaped `_` character.

A [left-flanking delimiter run](#) is a [delimiter run](#) that is (1) not followed by [Unicode whitespace](#), and either (2a) not followed by a [punctuation character](#), or (2b) followed by a [punctuation character](#) and preceded by [Unicode whitespace](#) or a [punctuation character](#). For purposes of this definition, the beginning and the end of the line count as Unicode whitespace.

A [right-flanking delimiter run](#) is a [delimiter run](#) that is (1) not preceded by [Unicode whitespace](#), and either (2a) not preceded by a [punctuation character](#), or (2b) preceded by a [punctuation character](#) and followed by [Unicode whitespace](#) or a [punctuation character](#). For purposes of this definition, the beginning and the end of the line count as Unicode whitespace.

Here are some examples of delimiter runs.

- left-flanking but not right-flanking:

```
***abc
 _abc
**"abc"
 _"abc"
```

- right-flanking but not left-flanking:

```
abc***
abc_
"abc"***
"abc" _
```

- Both left and right-flanking:

```
abc***def
"abc" _ "def"
```

- Neither left nor right-flanking:

```
abc *** def
a _ b
```

(The idea of distinguishing left-flanking and right-flanking delimiter runs based on the character before and the character after comes from Roopesh Chander's [vfmd](#). vfmd uses the terminology "emphasis indicator string" instead of "delimiter run," and its rules for distinguishing left- and right-flanking runs are a bit more complex than the ones given here.)

The following rules define emphasis and strong emphasis:

1. A single \* character [can open emphasis](#) iff (if and only if) it is part of a [left-flanking delimiter run](#).
2. A single \_ character [can open emphasis](#) iff it is part of a [left-flanking delimiter run](#) and either (a) not part of a [right-flanking delimiter run](#) or (b) part of a [right-flanking delimiter run](#) preceded by punctuation.
3. A single \* character [can close emphasis](#) iff it is part of a [right-flanking delimiter run](#).
4. A single \_ character [can close emphasis](#) iff it is part of a [right-flanking delimiter run](#) and either (a) not part of a [left-flanking delimiter run](#) or (b) part of a [left-flanking delimiter run](#) followed by punctuation.
5. A double \*\* [can open strong emphasis](#) iff it is part of a [left-flanking delimiter run](#).
6. A double \_\_ [can open strong emphasis](#) iff it is part of a [left-flanking delimiter run](#) and either (a) not part of a [right-flanking delimiter run](#) or (b) part of a [right-flanking delimiter run](#) preceded by punctuation.
7. A double \*\* [can close strong emphasis](#) iff it is part of a [right-flanking delimiter run](#).
8. A double \_\_ [can close strong emphasis](#) iff it is part of a [right-flanking delimiter run](#) and either (a) not part of a [left-flanking delimiter run](#) or (b) part of a [left-flanking delimiter run](#) followed by punctuation.
9. Emphasis begins with a delimiter that [can open emphasis](#) and ends with a delimiter that [can close emphasis](#), and that uses the same character ( \_ or \* ) as the opening delimiter. The opening and closing delimiters must belong to separate [delimiter runs](#). If one of the delimiters can both open and close emphasis, then the sum of the lengths of the delimiter runs containing the opening and closing delimiters must not be a multiple of 3 unless both lengths are multiples of 3.
10. Strong emphasis begins with a delimiter that [can open strong emphasis](#) and ends with a delimiter that [can close strong emphasis](#), and that uses the same character ( \_ or \* ) as the opening delimiter. The opening and closing delimiters must belong to separate [delimiter runs](#). If one of the delimiters can both open and close strong emphasis, then the sum of the lengths of the delimiter runs containing the opening and closing delimiters must not be a multiple of 3 unless both lengths are multiples of 3.
11. A literal \* character cannot occur at the beginning or end of \*-delimited emphasis or \*\*-delimited strong emphasis, unless it is backslash-escaped.
12. A literal \_ character cannot occur at the beginning or end of \_-delimited emphasis or \_\_-delimited strong emphasis, unless it is backslash-escaped.

Where rules 1–12 above are compatible with multiple parsings, the following principles resolve ambiguity:

13. The number of nestings should be minimized. Thus, for example, an interpretation `<strong>...</strong>` is always preferred to `<em><em>...</em></em>`.
14. An interpretation `<em><strong>...</strong></em>` is always preferred to `<strong><em>...</em></strong>`.

15. When two potential emphasis or strong emphasis spans overlap, so that the second begins before the first ends and ends after the first ends, the first takes precedence. Thus, for example, `*foo _bar* baz_` is parsed as `<em>foo _bar</em> baz_` rather than `*foo <em>bar* baz</em>`.
16. When there are two potential emphasis or strong emphasis spans with the same closing delimiter, the shorter one (the one that opens later) takes precedence. Thus, for example, `**foo **bar baz**` is parsed as `**foo <strong>bar baz</strong>` rather than `<strong>foo **bar baz</strong>`.
17. Inline code spans, links, images, and HTML tags group more tightly than emphasis. So, when there is a choice between an interpretation that contains one of these elements and one that does not, the former always wins. Thus, for example, `*[foo*](bar)` is parsed as `*<a href="bar">foo*</a>` rather than as `<em>[foo</em>](bar)`.

These rules can be illustrated through a series of examples.

Rule 1:

[Example 350](#)

```
foo bar <p>foo bar</p>
```

This is not emphasis, because the opening `*` is followed by whitespace, and hence not part of a [left-flanking delimiter run](#):

[Example 351](#)

```
a *foo bar* <p>a *foo bar*</p>
```

This is not emphasis, because the opening `*` is preceded by an alphanumeric and followed by punctuation, and hence not part of a [left-flanking delimiter run](#):

[Example 352](#)

```
a"foo"* <p>a""foo"*</p>
```

Unicode nonbreaking spaces count as whitespace, too:

[Example 353](#)

```
* a * <p>* a *</p>
```

Intraword emphasis with `*` is permitted:

[Example 354](#)

```
foo*bar* <p>foobar</p>
```

[Example 355](#)

```
5*6*78 <p>5678</p>
```

Rule 2:

[Example 356](#)

`_foo bar_`

`<p><em>foo bar</em></p>`

This is not emphasis, because the opening `_` is followed by whitespace:

[Example 357](#)

`_foo bar_`

`<p>_foo bar_</p>`

This is not emphasis, because the opening `_` is preceded by an alphanumeric and followed by punctuation:

[Example 358](#)

`a_ "foo" _`

`<p>a_&quot;foo&quot;;_</p>`

Emphasis with `_` is not allowed inside words:

[Example 359](#)

`foo_bar_`

`<p>foo_bar_</p>`

[Example 360](#)

`5_6_78`

`<p>5_6_78</p>`

[Example 361](#)

`пристаням_стремятся_`

`<p>пристаням_стремятся_</p>`

Here `_` does not generate emphasis, because the first delimiter run is right-flanking and the second left-flanking:

[Example 362](#)

`aa_"bb"_cc`

`<p>aa_&quot;bb&quot;;_cc</p>`

This is emphasis, even though the opening delimiter is both left- and right-flanking, because it is preceded by punctuation:

[Example 363](#)

`foo_(bar)_`

`<p>foo-<em>(bar)</em></p>`

Rule 3:

This is not emphasis, because the closing delimiter does not match the opening delimiter:

[Example 364](#)

`_foo*`

`<p>_foo*</p>`

This is not emphasis, because the closing `*` is preceded by whitespace:

#### [Example 365](#)

<code>*foo bar *</code>	<code>&lt;p&gt;*foo bar *&lt;/p&gt;</code>
-------------------------	--------------------------------------------

A newline also counts as whitespace:

#### [Example 366](#)

<code>*foo bar</code> <code>*</code>	<code>&lt;p&gt;*foo bar</code> <code>*&lt;/p&gt;</code>
-----------------------------------------	------------------------------------------------------------

This is not emphasis, because the second `*` is preceded by punctuation and followed by an alphanumeric (hence it is not part of a [right-flanking delimiter run](#)):

#### [Example 367](#)

<code>*(foo)</code>	<code>&lt;p&gt;*(foo)&lt;/p&gt;</code>
---------------------	----------------------------------------

The point of this restriction is more easily appreciated with this example:

#### [Example 368](#)

<code>*(foo)*</code>	<code>&lt;p&gt;&lt;em&gt;(&lt;em&gt;foo&lt;/em&gt;&lt;/em&gt;&lt;/p&gt;</code>
----------------------	--------------------------------------------------------------------------------

Intraword emphasis with `*` is allowed:

#### [Example 369](#)

<code>*foo*bar</code>	<code>&lt;p&gt;&lt;em&gt;foo&lt;/em&gt;bar&lt;/p&gt;</code>
-----------------------	-------------------------------------------------------------

Rule 4:

This is not emphasis, because the closing `_` is preceded by whitespace:

#### [Example 370](#)

<code>_foo bar _</code>	<code>&lt;p&gt;_foo bar _&lt;/p&gt;</code>
-------------------------	--------------------------------------------

This is not emphasis, because the second `_` is preceded by punctuation and followed by an alphanumeric:

#### [Example 371](#)

<code>_(foo)</code>	<code>&lt;p&gt;_(foo)&lt;/p&gt;</code>
---------------------	----------------------------------------

This is emphasis within emphasis:

#### [Example 372](#)

<code>_(foo)_</code>	<code>&lt;p&gt;&lt;em&gt;(&lt;em&gt;foo&lt;/em&gt;&lt;/em&gt;&lt;/p&gt;</code>
----------------------	--------------------------------------------------------------------------------

Intraword emphasis is disallowed for `_`:

### Example 373

<p>\_foo\_bar</p>

### Example 374

<p>\_п р и с т а н я м \_с т р е м я т с я</p>  
>

### Example 375

*foo\_bar\_baz*

This is emphasis, even though the closing delimiter is both left- and right-flanking, because it is followed by punctuation:

### Example 376

(bar).

Rule 5:

### Example 377

<p><strong>foo bar</strong></p>

This is not strong emphasis, because the opening delimiter is followed by whitespace:

### Example 378

**\*\* foo bar\*\***

This is not strong emphasis, because the opening **\*\*** is preceded by an alphanumeric and followed by punctuation, and hence not part of a [left-flanking delimiter run](#):

### Example 379

a\*\*&quot;foo&quot;\*\*</p>

Intraword strong emphasis with \*\* is permitted:

### Example 380

foo**bar**

Rule 6:

### Example 381

<p><strong>foo bar</strong></p>

This is not strong emphasis, because the opening delimiter is followed by whitespace:

### Example 382



\_\_foo bar\_\_

<p>\_\_foo bar\_\_</p>

A newline counts as whitespace:

[Example 383](#)

\_\_  
foo bar\_\_

<p>\_\_  
foo bar\_\_</p>

This is not strong emphasis, because the opening \_\_ is preceded by an alphanumeric and followed by punctuation:

[Example 384](#)

a\_\_"foo"\_\_

<p>a\_\_&quot;foo&quot;\_\_</p>

Intraword strong emphasis is forbidden with \_\_:

[Example 385](#)

foo\_\_bar\_\_

<p>foo\_\_bar\_\_</p>

[Example 386](#)

5\_\_6\_\_78

<p>5\_\_6\_\_78</p>

[Example 387](#)

п р и с т а н я м \_\_ с т р е м я т с я \_\_

<p>п р и с т а н я м \_\_ с т р е м я т с я \_\_</p>

[Example 388](#)

\_\_foo, • \_\_bar\_\_, baz\_\_

<p><strong>foo, <strong>bar</strong>, •  
baz</strong></p>

This is strong emphasis, even though the opening delimiter is both left- and right-flanking, because it is preceded by punctuation:

[Example 389](#)

foo-\_\_(bar)\_\_

<p>foo-<strong>(bar)</strong></p>

Rule 7:

This is not strong emphasis, because the closing delimiter is preceded by whitespace:

[Example 390](#)

\*\*foo bar \*\*

<p>\*\*foo bar \*\*</p>

(Nor can it be interpreted as an emphasized \*foo bar \*, because of Rule 11.)

This is not strong emphasis, because the second **\*\*** is preceded by punctuation and followed by an alphanumeric:

[Example 391](#)

<b>**(**foo)</b>	<code>&lt;p&gt;<b>**(**foo)</b>&lt;/p&gt;</code>
------------------	--------------------------------------------------

The point of this restriction is more easily appreciated with these examples:

[Example 392](#)

<b>*(**foo**)*</b>	<code>&lt;p&gt;&lt;em&gt;(&lt;strong&gt;foo&lt;/strong&gt;&lt;/em&gt;&lt;/p&gt;</code>
--------------------	----------------------------------------------------------------------------------------

[Example 393](#)

<b>**Gomphocarpus •(*Gomphocarpus physocarpus*, • syn. *Asclepias physocarpa*)**</b>	<code>&lt;p&gt;&lt;strong&gt;Gomphocarpus •(&lt;em&gt;Gomphocarpus • physocarpus&lt;/em&gt;, •syn. &lt;em&gt;Asclepias physocarpa&lt;/em&gt;)&lt;/strong&gt;&lt;/p&gt;</code>
------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

[Example 394](#)

<b>**foo “*bar*” foo**</b>	<code>&lt;p&gt;&lt;strong&gt;foo &amp;quot;&lt;em&gt;bar&lt;/em&gt;&amp;quot; • foo&lt;/strong&gt;&lt;/p&gt;</code>
----------------------------	-------------------------------------------------------------------------------------------------------------------------

Intraword emphasis:

[Example 395](#)

<b>**foo**bar</b>	<code>&lt;p&gt;&lt;strong&gt;foo&lt;/strong&gt;bar&lt;/p&gt;</code>
-------------------	---------------------------------------------------------------------

Rule 8:

This is not strong emphasis, because the closing delimiter is preceded by whitespace:

[Example 396](#)

<b>__foo bar •__</b>	<code>&lt;p&gt;__foo bar •__&lt;/p&gt;</code>
----------------------	-----------------------------------------------

This is not strong emphasis, because the second **\_\_** is preceded by punctuation and followed by an alphanumeric:

[Example 397](#)

<b>__(__foo)</b>	<code>&lt;p&gt;__(__foo)&lt;/p&gt;</code>
------------------	-------------------------------------------

The point of this restriction is more easily appreciated with this example:

[Example 398](#)

<b>_(__foo__)</b>	<code>&lt;p&gt;&lt;em&gt;(&lt;strong&gt;foo&lt;/strong&gt;&lt;/em&gt;&lt;/p&gt;</code>
-------------------	----------------------------------------------------------------------------------------

Intraword strong emphasis is forbidden with **\_\_**:

[Example 399](#)

\_\_foo\_\_bar

<p>\_\_foo\_\_bar</p>

#### [Example 400](#)

\_\_п р и с т а н я м\_\_с т р е м я т с я

<p>\_\_п р и с т а н я м\_\_с т р е м я т с я</p>

#### [Example 401](#)

\_\_foo\_\_bar\_\_baz\_\_

<p><strong>foo\_\_bar\_\_baz</strong></p>

This is strong emphasis, even though the closing delimiter is both left- and right-flanking, because it is followed by punctuation:

#### [Example 402](#)

\_\_(bar)\_\_.

<p><strong>(bar)</strong>.</p>

#### Rule 9:

Any nonempty sequence of inline elements can be the contents of an emphasized span.

#### [Example 403](#)

\*foo [bar] (/url)\*

<p><em>foo <a href="/url">bar</a></em></p>

#### [Example 404](#)

\*foo  
bar\*

<p><em>foo  
bar</em></p>

In particular, emphasis and strong emphasis can be nested inside emphasis:

#### [Example 405](#)

\_foo •\_bar\_ baz\_

<p><em>foo <strong>bar</strong> baz</em></p>

#### [Example 406](#)

\_foo •\_bar\_ baz\_

<p><em>foo <em>bar</em> baz</em></p>

#### [Example 407](#)

\_\_foo\_ bar\_

<p><em><em>foo</em> bar</em></p>

#### [Example 408](#)

\*foo \*bar\*\*

<p><em>foo <em>bar</em></em></p>

#### [Example 409](#)

\*foo \*\*bar\*\* baz\*

<p><em>foo <strong>bar</strong> baz</em></p>

#### [Example 410](#)

\*foo\*\*bar\*\*baz\*

<p><em>foo<strong>bar</strong>baz</em></p>

Note that in the preceding case, the interpretation

```
<p>foobarbaz</p>
```

is precluded by the condition that a delimiter that can both open and close (like the `*` after `foo`) cannot form emphasis if the sum of the lengths of the delimiter runs containing the opening and closing delimiters is a multiple of 3 unless both lengths are multiples of 3.

For the same reason, we don't get two consecutive emphasis sections in this example:

#### [Example 411](#)

```
*foo**bar* <p>foo**bar</p>
```

The same condition ensures that the following cases are all strong emphasis nested inside emphasis, even when the interior spaces are omitted:

#### [Example 412](#)

```
***foo** bar* <p>foo bar</p>
```

#### [Example 413](#)

```
*foo **bar*** <p>foo bar</p>
```

#### [Example 414](#)

```
*foo**bar*** <p>foobar</p>
```

When the lengths of the interior closing and opening delimiter runs are *both* multiples of 3, though, they can match to create emphasis:

#### [Example 415](#)

```
foo***bar***baz <p>foobarbaz</p>
```

#### [Example 416](#)

```
foo*****bar*****baz <p>foobar</p>
```

Indefinite levels of nesting are possible:

#### [Example 417](#)

```
*foo **bar *baz* bim** bop* <p>foo bar baz •
bim bop</p>
```

#### [Example 418](#)

```
*foo [*bar*](url)* <p>foo bar</p>
```

There can be no empty emphasis or strong emphasis:

#### [Example 419](#)

***\*\* is not an empty emphasis***

`<p>** is not an empty emphasis</p>`

#### [Example 420](#)

***\*\*\* is not an empty strong emphasis***

`<p>*** is not an empty strong emphasis</p>`

#### Rule 10:

Any nonempty sequence of inline elements can be the contents of an strongly emphasized span.

#### [Example 421](#)

***\*\*foo [bar] (/url)\*\****

`<p><strong>foo <a href="/url">bar</a>  
</strong></p>`

#### [Example 422](#)

***foo  
bar***

`<p><strong>foo  
bar</strong></p>`

In particular, emphasis and strong emphasis can be nested inside strong emphasis:

#### [Example 423](#)

***\_\_foo •\_bar\_ baz\_\_***

`<p><strong>foo <em>bar</em> baz</strong></p>`

#### [Example 424](#)

***\_\_foo •\_bar\_ baz\_\_***

`<p><strong>foo <strong>bar</strong> •  
baz</strong></p>`

#### [Example 425](#)

***\_\_\_\_foo\_\_ bar\_\_***

`<p><strong><strong>foo</strong> bar</strong>  
</p>`

#### [Example 426](#)

***\*\*foo \*\*bar\*\*\****

`<p><strong>foo <strong>bar</strong></strong>  
</p>`

#### [Example 427](#)

***\*\*foo \*bar\* baz\*\****

`<p><strong>foo <em>bar</em> baz</strong></p>`

#### [Example 428](#)

***\*\*foo\*bar\*baz\*\****

`<p><strong>foo<em>bar</em>baz</strong></p>`

#### [Example 429](#)

***\*\*\*foo\* bar\*\****

`<p><strong><em>foo</em> bar</strong></p>`

#### [Example 430](#)

**foo bar**

<p><strong>foo <em>bar</em></strong></p>

Indefinite levels of nesting are possible:

#### [Example 431](#)

**foo bar baz**  
bim bop

<p><strong>foo <em>bar <strong>baz</strong>  
bim</em> bop</strong></p>

#### [Example 432](#)

**foo [bar] (/url)**

<p><strong>foo <a href="/url"><em>bar</em>  
</a></strong></p>

There can be no empty emphasis or strong emphasis:

#### [Example 433](#)

**\_** is not an empty emphasis

<p>\_ is not an empty emphasis</p>

#### [Example 434](#)

**\_\_** is not an empty strong emphasis

<p>\_\_ is not an empty strong emphasis</p>

Rule 11:

#### [Example 435](#)

foo \*\*\*

<p>foo \*\*\*</p>

#### [Example 436](#)

foo \*\\*\*

<p>foo <em>\*</em></p>

#### [Example 437](#)

foo \*\_\*

<p>foo <em>\_</em></p>

#### [Example 438](#)

foo \*\*\*\*

<p>foo \*\*\*\*</p>

#### [Example 439](#)

foo \*\*\\*\*\*

<p>foo <strong>\*</strong></p>

#### [Example 440](#)

foo \*\*\_\*\*

<p>foo <strong>\_</strong></p>

Note that when delimiters do not match evenly, Rule 11 determines that the excess literal \* characters will appear outside of the emphasis, rather than inside it:

#### [Example 441](#)

**foo**

<p>\*<em>foo</em></p>

#### [Example 442](#)

**foo**

<p><em>foo</em>\*</p>

#### [Example 443](#)

**foo**

<p>\*<strong>foo</strong></p>

#### [Example 444](#)

**foo**

<p>\*\*\*<em>foo</em></p>

#### [Example 445](#)

**foo**

<p><strong>foo</strong>\*</p>

#### [Example 446](#)

**foo**

<p><em>foo</em>\*\*\*</p>

#### Rule 12:

#### [Example 447](#)

foo •

<p>foo •</p>

#### [Example 448](#)

foo •\

<p>foo <em>\_</em></p>

#### [Example 449](#)

foo •\*\_

<p>foo <em>\*</em></p>

#### [Example 450](#)

foo •

<p>foo •</p>

#### [Example 451](#)

foo •\

<p>foo <strong>\_</strong></p>

#### [Example 452](#)

foo •\*\_

<p>foo <strong>\*</strong></p>

#### [Example 453](#)

\_foo\_

<p>\_<em>foo</em></p>

Note that when delimiters do not match evenly, Rule 12 determines that the excess literal \_ characters will appear outside of the emphasis, rather than inside it:

#### [Example 454](#)

foo

<p><em>foo</em>\_</p>

#### [Example 455](#)

foo

<p>\_<strong>foo</strong></p>

#### [Example 456](#)

foo

<p>\_\_<em>foo</em></p>

#### [Example 457](#)

foo

<p><strong>foo</strong>\_</p>

#### [Example 458](#)

foo

<p><em>foo</em>\_\_</p>

Rule 13 implies that if you want emphasis nested directly inside emphasis, you must use different delimiters:

#### [Example 459](#)

**foo**

<p><strong>foo</strong></p>

#### [Example 460](#)

*foo*

<p><em><em>foo</em></em></p>

#### [Example 461](#)

foo

<p><strong>foo</strong></p>

#### [Example 462](#)

*foo*

<p><em><em>foo</em></em></p>

However, strong emphasis within strong emphasis is possible without switching delimiters:

#### [Example 463](#)

**foo**

<p><strong><strong>foo</strong></strong></p>

#### [Example 464](#)

foo

<p><strong><strong>foo</strong></strong></p>

Rule 13 can be applied to arbitrarily long sequences of delimiters:

#### [Example 465](#)

**foo**

<p><strong><strong><strong>foo</strong></strong></strong></p>

Rule 14:



#### [Example 466](#)

\*\*\*foo\*\*\*

<p><em><strong>foo</strong></em></p>

#### [Example 467](#)

\_\_\_\_foo\_\_\_\_

<p><em><strong><strong>foo</strong></strong></em></p>

Rule 15:

#### [Example 468](#)

\*foo \_bar\* baz\_

<p><em>foo \_bar</em> baz\_</p>

#### [Example 469](#)

\*foo \_bar \*baz bim\_\_ bam\*

<p><em>foo <strong>bar \*baz bim\_\_ bam</strong> •  
bam</em></p>

Rule 16:

#### [Example 470](#)

\*\*foo \*\*bar baz\*\*

<p>\*\*foo <strong>bar baz</strong></p>

#### [Example 471](#)

\*foo \*bar baz\*

<p>\*foo <em>bar baz</em></p>

Rule 17:

#### [Example 472](#)

\*[bar\*] (/url)

<p>\*<a href="/url">bar\*</a></p>

#### [Example 473](#)

\_foo [bar\_] (/url)

<p>\_foo <a href="/url">bar\_</a></p>

#### [Example 474](#)

\*

<p>\*</p>

#### [Example 475](#)

\*\*<a href="\*\*">

<p>\*\*<a href="\*\*"></p>

#### [Example 476](#)

\_\_<a href="\_\_">

<p>\_\_<a href="\_\_"></p>

#### [Example 477](#)

\*a `\*`\*

<p><em>a <code>\*</code></em></p>

#### [Example 478](#)

*a*

```
a<http://foo.bar/?q=>
```

<p>\*\*a<a href="http://foo.bar/?q=">http://foo.bar/?q=\*\*</a></p>

a<http://foo.bar/?q= >

```
<p>__a<a href="http://foo.bar/?q= "http://foo.bar/?q= </p>
```

A link contains [link text](#) (the visible text), a [link destination](#) (the URI that is the link destination), and optionally a [link title](#). There are two basic kinds of links in Markdown. In [inline links](#) the destination and title are given immediately after the link text. In [reference links](#) the destination and title are defined elsewhere in the document.

A [link text](#) consists of a sequence of zero or more inline elements enclosed by square brackets ([ and ]). The following rules apply:

- Links may not contain other links, at any level of nesting. If multiple otherwise valid link definitions appear nested inside each other, the inner-most definition is used.
- Brackets are allowed in the [link text](#) only if (a) they are backslash-escaped or (b) they appear as a matched pair of brackets, with an open bracket `[`, a sequence of zero or more inlines, and a close bracket `]`.
- Backtick [code spans](#), [autolinks](#), and raw [HTML tags](#) bind more tightly than the brackets in link text. Thus, for example, `[foo`]` could not be a link text, since the second `]` is part of a code span.
- The brackets in link text bind more tightly than markers for [emphasis and strong emphasis](#). Thus, for example, `*[foo*](url)` is a link.

A **link destination** consists of either

- a sequence of zero or more characters between an opening `<` and a closing `>` that contains no line breaks or unescaped `<` or `>` characters, or
- a nonempty sequence of characters that does not start with `<`, does not include ASCII space or control characters, and includes parentheses only if (a) they are backslash-escaped or (b) they are part of a balanced pair of unescaped parentheses. (Implementations may impose limits on parentheses nesting to avoid performance issues, but at least three levels of nesting should be supported.)

A **link title** consists of either

- a sequence of zero or more characters between straight double-quote characters ("), including a " character only if it is backslash-escaped, or
- a sequence of zero or more characters between straight single-quote characters ('), including a ' character only if it is backslash-escaped, or
- a sequence of zero or more characters between matching parentheses ((...)), including a ( or ) character only if it is backslash-escaped.

Although **link titles** may span multiple lines, they may not contain a **blank line**.

An [inline link](#) consists of a [link text](#) followed immediately by a left parenthesis (, optional [whitespace](#), an optional [link destination](#), an optional [link title](#) separated from the link destination by [whitespace](#), optional [whitespace](#), and a right parenthesis ). The link's text consists of the inlines contained in the [link text](#) (excluding the enclosing square brackets). The link's URI consists of the link destination, excluding enclosing `<...>` if present, with backslash-escapes in effect as described above. The link's title consists of the link title, excluding its enclosing delimiters, with backslash-escapes in effect as described above.

Here is a simple inline link:

#### [Example 481](#)

<code>[link] (/uri "title")</code>	<code>&lt;p&gt;&lt;a href="/uri" title="title"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
------------------------------------	-----------------------------------------------------------------------------------

The title may be omitted:

#### [Example 482](#)

<code>[link] (/uri)</code>	<code>&lt;p&gt;&lt;a href="/uri"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
----------------------------	---------------------------------------------------------------------

Both the title and the destination may be omitted:

#### [Example 483](#)

<code>[link] ()</code>	<code>&lt;p&gt;&lt;a href=""&gt;link&lt;/a&gt;&lt;/p&gt;</code>
------------------------	-----------------------------------------------------------------

#### [Example 484](#)

<code>[link] (&lt;&gt;)</code>	<code>&lt;p&gt;&lt;a href=""&gt;link&lt;/a&gt;&lt;/p&gt;</code>
--------------------------------	-----------------------------------------------------------------

The destination can only contain spaces if it is enclosed in pointy brackets:

#### [Example 485](#)

<code>[link] (/my uri)</code>	<code>&lt;p&gt;[link] (/my uri)&lt;/p&gt;</code>
-------------------------------	--------------------------------------------------

#### [Example 486](#)

<code>[link] (&lt;/my uri&gt;)</code>	<code>&lt;p&gt;&lt;a href="/my%20uri"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
---------------------------------------	--------------------------------------------------------------------------

The destination cannot contain line breaks, even if enclosed in pointy brackets:

#### [Example 487](#)

<code>[link] (foo bar)</code>	<code>&lt;p&gt;[link] (foo bar)&lt;/p&gt;</code>
-----------------------------------	------------------------------------------------------

#### [Example 488](#)

<code>[link] (&lt;foo bar&gt;)</code>	<code>&lt;p&gt;[link] (&lt;foo bar&gt;)&lt;/p&gt;</code>
-------------------------------------------	--------------------------------------------------------------

The destination can contain ) if it is enclosed in pointy brackets:

#### [Example 489](#)

[a] (<b>c>)

<p><a href="b">a</a></p>

Pointy brackets that enclose links must be unescaped:

#### [Example 490](#)

[link] (<foo>)

<p>[link] (&lt;foo&gt;)</p>

These are not links, because the opening pointy bracket is not matched properly:

#### [Example 491](#)

[a] (<b>c  
[a] (<b>c>  
[a] (<b>c)

<p>[a] (&lt;b>c  
[a] (&lt;b>c&gt;  
[a] (<b>c)</p>

Parentheses inside the link destination may be escaped:

#### [Example 492](#)

[link] (\(foo\))

<p><a href="(foo)">link</a></p>

Any number of parentheses are allowed without escaping, as long as they are balanced:

#### [Example 493](#)

[link] (foo(and(bar)))

<p><a href="foo(and(bar))">link</a></p>

However, if you have unbalanced parentheses, you need to escape or use the <...> form:

#### [Example 494](#)

[link] (foo\ (and\ (bar\))

<p><a href="foo (and (bar)">link</a></p>

#### [Example 495](#)

[link] (<foo(and(bar)>)

<p><a href="foo(and(bar)">link</a></p>

Parentheses and other symbols can also be escaped, as usual in Markdown:

#### [Example 496](#)

[link] (foo\)\:)

<p><a href="foo):">link</a></p>

A link can contain fragment identifiers and queries:

#### [Example 497](#)

<code>[link] (#fragment)</code>	<code>&lt;p&gt;&lt;a href="#fragment"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
<code>[link] (http://example.com#fragment)</code>	<code>&lt;p&gt;&lt;a href="http://example.com#fragment"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
<code>[link] (http://example.com?foo=3#frag)</code>	<code>&lt;p&gt;&lt;a href="http://example.com?foo=3#frag"&gt;link&lt;/a&gt;&lt;/p&gt;</code>

Note that a backslash before a non-escapable character is just a backslash:

#### [Example 498](#)

<code>[link] (foo\bar)</code>	<code>&lt;p&gt;&lt;a href="foo%5Cbar"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
-------------------------------	--------------------------------------------------------------------------

URL-escaping should be left alone inside the destination, as all URL-escaped characters are also valid URL characters. Entity and numerical character references in the destination will be parsed into the corresponding Unicode code points, as usual. These may be optionally URL-escaped when written as HTML, but this spec does not enforce any particular policy for rendering URLs in HTML or other formats. Renderers may make different decisions about how to escape or normalize URLs in the output.

#### [Example 499](#)

<code>[link] (foo%20b&amp;auml;)</code>	<code>&lt;p&gt;&lt;a href="foo%20b%C3%A4"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
-----------------------------------------	------------------------------------------------------------------------------

Note that, because titles can often be parsed as destinations, if you try to omit the destination and keep the title, you'll get unexpected results:

#### [Example 500](#)

<code>[link] ("title")</code>	<code>&lt;p&gt;&lt;a href="%22title%22"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
-------------------------------	----------------------------------------------------------------------------

Titles may be in single quotes, double quotes, or parentheses:

#### [Example 501](#)

<code>[link] (/url "title")</code>	<code>&lt;p&gt;&lt;a href="/url" title="title"&gt;link&lt;/a&gt;</code>
<code>[link] (/url 'title')</code>	<code>&lt;a href="/url" title="title"&gt;link&lt;/a&gt;</code>
<code>[link] (/url (title))</code>	<code>&lt;a href="/url" title="title"&gt;link&lt;/a&gt;&lt;/p&gt;</code>

Backslash escapes and entity and numeric character references may be used in titles:

#### [Example 502](#)

<code>[link] (/url "title \&amp;quot;")</code>	<code>&lt;p&gt;&lt;a href="/url" title="title • &amp;quot;&amp;quot;"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
------------------------------------------------	--------------------------------------------------------------------------------------------------------------

Titles must be separated from the link using a [whitespace](#). Other [Unicode whitespace](#) like non-breaking space doesn't work.

#### [Example 503](#)

<code>[link] (/url "title")</code>	<code>&lt;p&gt;&lt;a href="/url%C2%A0%22title%22"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
------------------------------------	--------------------------------------------------------------------------------------

Nested balanced quotes are not allowed without escaping:

#### [Example 504](#)

<code>[link] (/url "title "and" title")</code>	<code>&lt;p&gt;[link] (/url &amp;quot;title &amp;quot;and&amp;quot;; • title&amp;quot;)&lt;/p&gt;</code>
------------------------------------------------	----------------------------------------------------------------------------------------------------------

But it is easy to work around this by using a different quote type:

#### [Example 505](#)

<code>[link] (/url 'title "and" title')</code>	<code>&lt;p&gt;&lt;a href="/url" title="title • &amp;quot;and&amp;quot;; title"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
------------------------------------------------	--------------------------------------------------------------------------------------------------------------------

(Note: Markdown.pl did allow double quotes inside a double-quoted title, and its test suite included a test demonstrating this. But it is hard to see a good rationale for the extra complexity this brings, since there are already many ways—backslash escaping, entity and numeric character references, or using a different quote type for the enclosing title—to write titles containing double quotes. Markdown.pl's handling of titles has a number of other strange features. For example, it allows single-quoted titles in inline links, but not reference links. And, in reference links but not inline links, it allows a title to begin with " and end with ). Markdown.pl 1.0.1 even allows titles with no closing quotation mark, though 1.0.2b8 does not. It seems preferable to adopt a simple, rational rule that works the same way in inline links and link reference definitions.)

[Whitespace](#) is allowed around the destination and title:

#### [Example 506](#)

<code>[link] ( ••/uri ••"title" ••)</code>	<code>&lt;p&gt;&lt;a href="/uri" title="title"&gt;link&lt;/a&gt;&lt;/p&gt;</code>
--------------------------------------------	-----------------------------------------------------------------------------------

But it is not allowed between the link text and the following parenthesis:

#### [Example 507](#)

<code>[link] •(/uri)</code>	<code>&lt;p&gt;[link] •(/uri)&lt;/p&gt;</code>
-----------------------------	------------------------------------------------

The link text may contain balanced brackets, but not unbalanced ones, unless they are escaped:

#### [Example 508](#)

<code>[link [foo [bar]]] (/uri)</code>	<code>&lt;p&gt;&lt;a href="/uri"&gt;link [foo [bar]]&lt;/a&gt;&lt;/p&gt;</code>
----------------------------------------	---------------------------------------------------------------------------------

#### [Example 509](#)

<code>[link [bar] (/uri)</code>	<code>&lt;p&gt;[link [bar] (/uri)&lt;/p&gt;</code>
---------------------------------	----------------------------------------------------

#### [Example 510](#)

<code>[link [bar] (/uri)</code>	<code>&lt;p&gt;[link &lt;a href="/uri"&gt;bar&lt;/a&gt;&lt;/p&gt;</code>
---------------------------------	--------------------------------------------------------------------------

#### [Example 511](#)

<code>[link \[bar] (/uri)</code>	<code>&lt;p&gt;&lt;a href="/uri"&gt;link [bar&lt;/a&gt;&lt;/p&gt;</code>
----------------------------------	--------------------------------------------------------------------------

The link text may contain inline content:

#### [Example 512](#)

<code>[link *foo **bar** `#`*](/uri)</code>	<code>&lt;p&gt;&lt;a href="/uri"&gt;link &lt;em&gt;foo • &lt;strong&gt;bar&lt;/strong&gt; &lt;code&gt;#&lt;/code&gt;&lt;/em&gt;&lt;/a&gt; &lt;/p&gt;</code>
---------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------

#### [Example 513](#)

<code>[![moon](moon.jpg)](/uri)</code>	<code>&lt;p&gt;&lt;a href="/uri"&gt;&lt;img src="moon.jpg" • alt="moon" /&gt;&lt;/a&gt;&lt;/p&gt;</code>
----------------------------------------	--------------------------------------------------------------------------------------------------------------

However, links may not contain other links, at any level of nesting.

#### [Example 514](#)

<code>[foo [bar](/uri)](/uri)</code>	<code>&lt;p&gt;[foo &lt;a href="/uri"&gt;bar&lt;/a&gt;](/uri)&lt;/p&gt;</code>
--------------------------------------	--------------------------------------------------------------------------------

#### [Example 515](#)

<code>[foo *[bar [baz](/uri)](/uri)*](/uri)</code>	<code>&lt;p&gt;[foo &lt;em&gt;[bar &lt;a href="/uri"&gt;baz&lt;/a&gt;] (/uri)&lt;/em&gt;](/uri)&lt;/p&gt;</code>
----------------------------------------------------	----------------------------------------------------------------------------------------------------------------------

#### [Example 516](#)

<code>![[[foo](uri1)](uri2)](uri3)</code>	<code>&lt;p&gt;&lt;img src="uri3" alt="[foo](uri2)" /&gt;&lt;/p&gt;</code>
-------------------------------------------	----------------------------------------------------------------------------

These cases illustrate the precedence of link text grouping over emphasis grouping:

#### [Example 517](#)

<code>*[foo*](/uri)</code>	<code>&lt;p&gt;*&lt;a href="/uri"&gt;foo*&lt;/a&gt;&lt;/p&gt;</code>
----------------------------	----------------------------------------------------------------------

#### [Example 518](#)

<code>[foo *bar](baz*)</code>	<code>&lt;p&gt;&lt;a href="baz*"&gt;foo *bar&lt;/a&gt;&lt;/p&gt;</code>
-------------------------------	-------------------------------------------------------------------------

Note that brackets that *aren't* part of links do not take precedence:

#### [Example 519](#)

<code>*foo [bar* baz]</code>	<code>&lt;p&gt;&lt;em&gt;foo [bar&lt;/em&gt; baz]&lt;/p&gt;</code>
------------------------------	--------------------------------------------------------------------

These cases illustrate the precedence of HTML tags, code spans, and autolinks over link grouping:

#### [Example 520](#)

<code>[foo &lt;bar attr="](baz)"&gt;</code>	<code>&lt;p&gt;[foo &lt;bar attr="](baz)"&gt;&lt;/p&gt;</code>
---------------------------------------------	----------------------------------------------------------------

#### [Example 521](#)

<code>[foo`](/uri)`</code>	<code>&lt;p&gt;[foo&lt;code&gt;](/uri)&lt;/code&gt;&lt;/p&gt;</code>
----------------------------	----------------------------------------------------------------------

### Example 522

```
[foo<http://example.com/?search=](uri)> <p>[foo<a href="http://example.com/?
search=%5D(uri)">http://example.com/?
search=](uri)</p>
```

There are three kinds of [reference links](#): [full](#), [collapsed](#), and [shortcut](#).

A [full reference link](#) consists of a [link text](#) immediately followed by a [link label](#) that [matches](#) a [link reference definition](#) elsewhere in the document.

A [link label](#) begins with a left bracket ([) and ends with the first right bracket (]) that is not backslash-escaped. Between these brackets there must be at least one [non-whitespace character](#). Unescaped square bracket characters are not allowed inside the opening and closing square brackets of [link labels](#). A link label can have at most 999 characters inside the square brackets.

One label [matches](#) another just in case their normalized forms are equal. To normalize a label, strip off the opening and closing brackets, perform the *Unicode case fold*, strip leading and trailing [whitespace](#) and collapse consecutive internal [whitespace](#) to a single space. If there are multiple matching reference link definitions, the one that comes first in the document is used. (It is desirable in such cases to emit a warning.)

The contents of the first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching [link reference definition](#).

Here is a simple example:

### Example 523

```
[foo][bar] <p>foo</p>
[bar]: /url "title"
```

The rules for the [link text](#) are the same as with [inline links](#). Thus:

The link text may contain balanced brackets, but not unbalanced ones, unless they are escaped:

### Example 524

```
[link {foo {bar}}][ref] <p>link {foo {bar}}</p>
[ref]: /uri
```

### Example 525

```
[link \[bar][ref] <p>link [bar</p>
[ref]: /uri
```

The link text may contain inline content:

### Example 526

```
[link *foo **bar** `#`][ref] <p>link foo •
bar <code>#</code>
[ref]: /uri</p>
```



#### [Example 527](#)

[!moon]moon.jpg[ref]  
[ref]: /uri

```
<p></p>
```

However, links may not contain other links, at any level of nesting.

#### [Example 528](#)

[foo [bar] (/uri)] [ref]  
[ref]: /uri

```
<p>[foo bar]<a •
href="/uri">ref</p>
```

#### [Example 529](#)

[foo \*bar [baz] [ref] \*] [ref]  
[ref]: /uri

```
<p>[foo bar baz]
ref</p>
```

(In the examples above, we have two [shortcut reference links](#) instead of one [full reference link](#).)

The following cases illustrate the precedence of link text grouping over emphasis grouping:

#### [Example 530](#)

\*[foo\*] [ref]  
[ref]: /uri

```
<p>*foo*</p>
```

#### [Example 531](#)

[foo \*bar] [ref]  
[ref]: /uri

```
<p>foo *bar</p>
```

These cases illustrate the precedence of HTML tags, code spans, and autolinks over link grouping:

#### [Example 532](#)

[foo <bar attr="[ref]">  
[ref]: /uri

```
<p>[foo <bar attr="[ref]"></p>
```

#### [Example 533](#)

[foo`][ref]  
[ref]: /uri

```
<p>[foo<code>][ref]</code></p>
```

#### [Example 534](#)

[foo<http://example.com/?search=][ref]>  
[ref]: /uri

```
<p>[foo<a href="http://example.com/?
search=%5D%5Bref%5D">http://example.com/?
search=][ref]</p>
```

Matching is case-insensitive:

[Example 535](#)

```
[foo][BaR] <p>foo</p>
[bar]: /url "title"
```

Unicode case fold is used:

[Example 536](#)

```
[Т о л п о й][Т о л п о й] is a Russian • <p>Т о л п о й is a •
word. Russian word.</p>
[Т О Л П О Й]: /url
```

Consecutive internal [whitespace](#) is treated as one space for purposes of determining matching:

[Example 537](#)

```
[Foo
•bar]: /url <p>Baz</p>
[Baz][Foo bar]
```

No [whitespace](#) is allowed between the [link text](#) and the [link label](#):

[Example 538](#)

```
[foo] •[bar] <p>[foo] <a href="/url" •
[bar]: /url "title" title="title">bar</p>
```

[Example 539](#)

```
[foo] <p>[foo]
[bar] bar</p>
[bar]: /url "title"
```

This is a departure from John Gruber's original Markdown syntax description, which explicitly allows whitespace between the link text and the link label. It brings reference links in line with [inline links](#), which (according to both original Markdown and this spec) cannot have whitespace after the link text. More importantly, it prevents inadvertent capture of consecutive [shortcut reference links](#). If whitespace is allowed between the link text and the link label, then in the following we will have a single reference link, not two shortcut reference links, as intended:

```
[foo]
[bar]

[foo]: /url1
[bar]: /url2
```

(Note that [shortcut reference links](#) were introduced by Gruber himself in a beta version of Markdown.pl, but never included in the official syntax description. Without shortcut reference links, it is harmless to allow space between the link text and link label; but once shortcut

references are introduced, it is too dangerous to allow this, as it frequently leads to unintended results.)

When there are multiple matching [link reference definitions](#), the first is used:

#### [Example 540](#)

[foo]: ↗url1	<p><a href="/url1">bar</a></p>
[foo]: ↗url2	
[bar][foo]	

Note that matching is performed on normalized strings, not parsed inline content. So the following does not match, even though the labels define equivalent inline content:

#### [Example 541](#)

[bar][foo\!]	<p>[bar][foo!]</p>
[foo!]: ↗url	

[Link labels](#) cannot contain brackets, unless they are backslash-escaped:

#### [Example 542](#)

[foo][ref[]]	<p>[foo][ref[]]</p>
[ref[]]: ↗uri	<p>[ref[]]: ↗uri</p>

#### [Example 543](#)

[foo][ref[bar]]	<p>[foo][ref[bar]]</p>
[ref[bar]]: ↗uri	<p>[ref[bar]]: ↗uri</p>

#### [Example 544](#)

[[[foo]]]	<p>[[[foo]]]</p>
[[[foo]]]: ↗url	<p>[[[foo]]]: ↗url</p>

#### [Example 545](#)

[foo][ref\[]]	<p><a href="/uri">foo</a></p>
[ref\[]]: ↗uri	

Note that in this example `[]` is not backslash-escaped:

#### [Example 546](#)

[bar\\]: ↗uri	<p><a href="/uri">bar\\</a></p>
[bar\\]	

A [link label](#) must contain at least one [non-whitespace character](#):

#### [Example 547](#)

<code>[]</code>	<code>&lt;p&gt;[]&lt;/p&gt;</code>
<code>[]: ↗uri</code>	<code>&lt;p&gt;[]: ↗uri&lt;/p&gt;</code>

#### [Example 548](#)

<code>[</code>	<code>&lt;p&gt;[</code>
<code>•</code>	<code>]&lt;/p&gt;</code>
<code>[</code>	<code>&lt;p&gt;[</code>
<code>↗]: ↗uri</code>	<code>]: ↗uri&lt;/p&gt;</code>

A **collapsed reference link** consists of a [link label](#) that [matches](#) a [link reference definition](#) elsewhere in the document, followed by the string `[]`. The contents of the first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching reference link definition. Thus, `[foo] []` is equivalent to `[foo] [foo]`.

#### [Example 549](#)

<code>[foo] []</code>	<code>&lt;p&gt;&lt;a href="/url" title="title"&gt;foo&lt;/a&gt;&lt;/p&gt;</code>
<code>[foo]: ↗url "title"</code>	

#### [Example 550](#)

<code>[*foo* bar] []</code>	<code>&lt;p&gt;&lt;a href="/url" title="title"&gt;&lt;em&gt;foo&lt;/em&gt; •</code>
<code>[*foo* bar]: ↗url "title"</code>	<code>bar&lt;/a&gt;&lt;/p&gt;</code>

The link labels are case-insensitive:

#### [Example 551](#)

<code>[Foo] []</code>	<code>&lt;p&gt;&lt;a href="/url" title="title"&gt;Foo&lt;/a&gt;&lt;/p&gt;</code>
<code>[foo]: ↗url "title"</code>	

As with full reference links, [whitespace](#) is not allowed between the two sets of brackets:

#### [Example 552](#)

<code>[foo] •</code>	<code>&lt;p&gt;&lt;a href="/url" title="title"&gt;foo&lt;/a&gt;</code>
<code>[]</code>	<code>&lt;/p&gt;</code>
<code>[foo]: ↗url "title"</code>	

A **shortcut reference link** consists of a [link label](#) that [matches](#) a [link reference definition](#) elsewhere in the document and is not followed by `[]` or a link label. The contents of the first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching link reference definition. Thus, `[foo]` is equivalent to `[foo] []`.

#### [Example 553](#)

[foo]	<p><a href="/url" title="title">foo</a></p>
[foo]: ↗url "title"	

#### Example 554

[*foo* bar]	<p><a href="/url" title="title"><em>foo</em> • bar</a></p>
[*foo* bar]: ↗url "title"	

#### Example 555

[[*foo* bar]]	<p>[<a href="/url" title="title"><em>foo</em> bar</a>]</p>
[*foo* bar]: ↗url "title"	

#### Example 556

[[bar <foo]]	<p>[[bar <a href="/url">foo</a></p>
[foo]: ↗url	

The link labels are case-insensitive:

#### Example 557

[Foo]	<p><a href="/url" title="title">Foo</a></p>
[foo]: ↗url "title"	

A space after the link text should be preserved:

#### Example 558

[foo] bar	<p><a href="/url">foo</a> bar</p>
[foo]: ↗url	

If you just want bracketed text, you can backslash-escape the opening bracket to avoid links:

#### Example 559

\[foo]	<p>[foo]</p>
[foo]: ↗url "title"	

Note that this is a link, because a link label ends with the first following closing bracket:

#### Example 560

[foo*]: ↗url	<p>*<a href="/url">foo*</a></p>
*[foo*]	

Full and compact references take precedence over shortcut references:

#### [Example 561](#)

[foo][bar]	<p><a href="/url2">foo</a></p>
[foo]: ↗url1	
[bar]: ↗url2	

#### [Example 562](#)

[foo][]	<p><a href="/url1">foo</a></p>
[foo]: ↗url1	

Inline links also take precedence:

#### [Example 563](#)

[foo]()	<p><a href="">foo</a></p>
[foo]: ↗url1	

#### [Example 564](#)

[foo](not a link)	<p><a href="/url1">foo</a>(not a link)</p>
[foo]: ↗url1	

In the following case [bar][baz] is parsed as a reference, [foo] as normal text:

#### [Example 565](#)

[foo][bar][baz]	<p>[foo]<a href="/url">bar</a></p>
[baz]: ↗url	

Here, though, [foo][bar] is parsed as a reference, since [bar] is defined:

#### [Example 566](#)

[foo][bar][baz]	<p><a href="/url2">foo</a><a href="/url1">baz</a></p>
[baz]: ↗url1	
[bar]: ↗url2	

Here [foo] is not parsed as a shortcut reference, because it is followed by a link label (even though [bar] is not defined):

#### [Example 567](#)

[foo][bar][baz]	<p>[foo]<a href="/url1">bar</a></p>
[baz]: ↗url1	
[foo]: ↗url2	

## 6.6 Images

Syntax for images is like the syntax for links, with one difference. Instead of [link text](#), we have an [image description](#). The rules for this are the same as for [link text](#), except that (a) an image description starts with `![` rather than `[`, and (b) an image description may contain links. An image description has inline elements as its contents. When an image is rendered to HTML, this is standardly used as the image's `alt` attribute.

#### [Example 568](#)

<code>![foo](/url "title")</code>	<code>&lt;p&gt;&lt;img src="/url" alt="foo" title="title" • &gt;&lt;/p&gt;</code>
-----------------------------------	---------------------------------------------------------------------------------------

#### [Example 569](#)

<code>![foo *bar*]</code>	<code>&lt;p&gt;&lt;img src="train.jpg" alt="foo bar" •</code>
<code>[foo *bar*]: train.jpg "train &amp; tracks"</code>	<code>title="train &amp; tracks" &gt;&lt;/p&gt;</code>

#### [Example 570](#)

<code>![foo ![bar](/url)](/url2)</code>	<code>&lt;p&gt;&lt;img src="/url2" alt="foo bar" &gt;&lt;/p&gt;</code>
-----------------------------------------	------------------------------------------------------------------------

#### [Example 571](#)

<code>![foo [bar](/url)](/url2)</code>	<code>&lt;p&gt;&lt;img src="/url2" alt="foo bar" &gt;&lt;/p&gt;</code>
----------------------------------------	------------------------------------------------------------------------

Though this spec is concerned with parsing, not rendering, it is recommended that in rendering to HTML, only the plain string content of the [image description](#) be used. Note that in the above example, the `alt` attribute's value is `foo bar`, not `foo [bar](/url)` or `foo <a href="/url">bar</a>`. Only the plain string content is rendered, without formatting.

#### [Example 572](#)

<code>![foo *bar*] []</code>	<code>&lt;p&gt;&lt;img src="train.jpg" alt="foo bar" •</code>
<code>[foo *bar*]: train.jpg "train &amp; tracks"</code>	<code>title="train &amp; tracks" &gt;&lt;/p&gt;</code>

#### [Example 573](#)

<code>![foo *bar*][foobar]</code>	<code>&lt;p&gt;&lt;img src="train.jpg" alt="foo bar" •</code>
<code>[FOOBAR]: train.jpg "train &amp; tracks"</code>	<code>title="train &amp; tracks" &gt;&lt;/p&gt;</code>

#### [Example 574](#)

<code>![foo](train.jpg)</code>	<code>&lt;p&gt;&lt;img src="train.jpg" alt="foo" &gt;&lt;/p&gt;</code>
--------------------------------	------------------------------------------------------------------------

#### [Example 575](#)

<code>My ![foo bar](/path/to/train.jpg • "title" ...)</code>	<code>&lt;p&gt;My &lt;img src="/path/to/train.jpg" alt="foo •</code>
	<code>bar" title="title" &gt;&lt;/p&gt;</code>

#### [Example 576](#)

<code>![foo](&lt;url&gt;)</code>	<code>&lt;p&gt;&lt;img src="url" alt="foo" &gt;&lt;/p&gt;</code>
----------------------------------	------------------------------------------------------------------

#### [Example 577](#)

<code></code>	<code>&lt;p&gt;&lt;img src="/url" alt="" &gt;&lt;/p&gt;</code>
------------------------	----------------------------------------------------------------

Reference-style:

[Example 578](#)

![foo][bar]

[bar]: ↗url

<p></p>

[Example 579](#)

![foo][bar]

[BAR]: ↗url

<p></p>

Collapsed:

[Example 580](#)

![foo][]

[foo]: ↗url "title"

<p></p>

[Example 581](#)

![\*foo\* bar][]

[\*foo\* bar]: ↗url "title"

<p></p>

The labels are case-insensitive:

[Example 582](#)

![Foo][]

[foo]: ↗url "title"

<p></p>

As with reference links, [whitespace](#) is not allowed between the two sets of brackets:

[Example 583](#)

![foo] •  
[]

[foo]: ↗url "title"

<p>  
[]</p>

Shortcut:

[Example 584](#)

![foo]

[foo]: ↗url "title"

<p></p>

[Example 585](#)



<code>![*foo* bar]</code>	<code>&lt;p&gt;&lt;img src="/url" alt="foo bar" •</code>
<code>[*foo* bar]: ↗url "title"</code>	<code>title="title" ↗&gt;&lt;/p&gt;</code>

Note that link labels cannot contain unescaped brackets:

#### [Example 586](#)

<code>![[foo]]</code>	<code>&lt;p&gt;![[foo]]&lt;/p&gt;</code>
<code>[[foo]]: ↗url "title"</code>	<code>&lt;p&gt;[[foo]]: ↗url &amp;quot;title&amp;quot;;&lt;/p&gt;</code>

The link labels are case-insensitive:

#### [Example 587](#)

<code>![Foo]</code>	<code>&lt;p&gt;&lt;img src="/url" alt="Foo" title="title" •</code>
<code>[foo]: ↗url "title"</code>	<code>/&gt;&lt;/p&gt;</code>

If you just want a literal `!` followed by bracketed text, you can backslash-escape the opening `[`:

#### [Example 588](#)

<code>!\[foo]</code>	<code>&lt;p&gt;!\[foo]&lt;/p&gt;</code>
<code>[foo]: ↗url "title"</code>	

If you want a link after a literal `!`, backslash-escape the `!`:

#### [Example 589](#)

<code>\![foo]</code>	<code>&lt;p&gt;!\&lt;a href="/url" title="title"&gt;foo&lt;/a&gt;&lt;/p&gt;</code>
<code>[foo]: ↗url "title"</code>	

## 6.7 Autolinks

**Autolinks** are absolute URIs and email addresses inside `<` and `>`. They are parsed as links, with the URL or email address as the link label.

A **URI autolink** consists of `<`, followed by an [absolute URI](#) followed by `>`. It is parsed as a link to the URI, with the URI as the link's label.

An **absolute URI**, for these purposes, consists of a [scheme](#) followed by a colon (`:`) followed by zero or more characters other than ASCII [whitespace](#) and control characters, `<`, and `>`. If the URI includes these characters, they must be percent-encoded (e.g. `%20` for a space).

For purposes of this spec, a **scheme** is any sequence of 2–32 characters beginning with an ASCII letter and followed by any combination of ASCII letters, digits, or the symbols plus (`+`), period (`.`), or hyphen (`-`).

Here are some valid autolinks:

#### [Example 590](#)

<http://foo.bar.baz>

<p><a href="http://foo.bar.baz">http://foo.bar.baz</a></p>

#### [Example 591](#)

<http://foo.bar.baz/test?q=hello&id=22&boolean>

<p><a href="http://foo.bar.baz/test?q=hello&id=22&boolean">http://foo.bar.baz/test?q=hello&id=22&boolean</a></p>

#### [Example 592](#)

<irc://foo.bar:2233/baz>

<p><a href="irc://foo.bar:2233/baz">irc://foo.bar:2233/baz</a></p>

Uppercase is also fine:

#### [Example 593](#)

<MAILTO:FOO@BAR.BAZ>

<p><a href="mailto:FOO@BAR.BAZ">MAILTO:FOO@BAR.BAZ</a></p>

Note that many strings that count as [absolute URIs](#) for purposes of this spec are not valid URIs, because their schemes are not registered or because of other problems with their syntax:

#### [Example 594](#)

<a+b+c:d>

<p><a href="a+b+c:d">a+b+c:d</a></p>

#### [Example 595](#)

<made-up-scheme://foo,bar>

<p><a href="made-up-scheme://foo,bar">made-up-scheme://foo,bar</a></p>

#### [Example 596](#)

<http://../>

<p><a href="http://../">http://../</a></p>

#### [Example 597](#)

<localhost:5001/foo>

<p><a href="localhost:5001/foo">localhost:5001/foo</a></p>

Spaces are not allowed in autolinks:

#### [Example 598](#)

<http://foo.bar/baz &#x201d>

<p>&#x201d;http://foo.bar/baz &#x201d;</p>

Backslash-escapes do not work inside autolinks:

#### [Example 599](#)

<http://example.com/\[\]>

<p><a •  
href="http://example.com/%5C%5B%5C">http://e  
xample.com/\[\</a></p>

An **email autolink** consists of <, followed by an **email address**, followed by >. The link's label is the email address, and the URL is `mailto:` followed by the email address.

An **email address**, for these purposes, is anything that matches the [non-normative regex from the HTML5 spec](#):

```
/^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$/
```

Examples of email autolinks:

#### [Example 600](#)

<foo@bar.example.com>

<p><a •  
href="mailto:foo@bar.example.com">foo@bar.ex  
ample.com</a></p>

#### [Example 601](#)

<foo+special@Bar.baz-bar0.com>

<p><a href="mailto:foo+special@Bar.baz-  
bar0.com">foo+special@Bar.baz-bar0.com</a>  
</p>

Backslash-escapes do not work inside email autolinks:

#### [Example 602](#)

<foo\+@bar.example.com>

<p>&lt;foo+@bar.example.com&gt;</p>

These are not autolinks:

#### [Example 603](#)

<>

<p>&lt;&gt;</p>

#### [Example 604](#)

< http://foo.bar ➤

<p>&lt; http://foo.bar &gt;</p>

#### [Example 605](#)

<m:abc>

<p>&lt;m:abc&gt;</p>

#### [Example 606](#)

<foo.bar.baz>

<p>&lt;foo.bar.baz&gt;</p>

#### [Example 607](#)

http://example.com

<p>http://example.com</p>

#### [Example 608](#)

foo@bar.example.com

<p>foo@bar.example.com</p>

## 6.8 Raw HTML

Text between < and > that looks like an HTML tag is parsed as a raw HTML tag and will be rendered in HTML without escaping. Tag and attribute names are not limited to current HTML tags, so custom tags (and even, say, DocBook tags) may be used.

Here is the grammar for tags:

A [tag name](#) consists of an ASCII letter followed by zero or more ASCII letters, digits, or hyphens (-).

An [attribute](#) consists of [whitespace](#), an [attribute name](#), and an optional [attribute value specification](#).

An [attribute name](#) consists of an ASCII letter, \_, or :, followed by zero or more ASCII letters, digits, \_, ., :, or -. (Note: This is the XML specification restricted to ASCII. HTML5 is laxer.)

An [attribute value specification](#) consists of optional [whitespace](#), a = character, optional [whitespace](#), and an [attribute value](#).

An [attribute value](#) consists of an [unquoted attribute value](#), a [single-quoted attribute value](#), or a [double-quoted attribute value](#).

An [unquoted attribute value](#) is a nonempty string of characters not including [whitespace](#), ", ', =, <, >, or `.

A [single-quoted attribute value](#) consists of ', zero or more characters not including ', and a final '.

A [double-quoted attribute value](#) consists of ", zero or more characters not including ", and a final ".

An [open tag](#) consists of a < character, a [tag name](#), zero or more [attributes](#), optional [whitespace](#), an optional / character, and a > character.

A [closing tag](#) consists of the string </, a [tag name](#), optional [whitespace](#), and the character >.

An [HTML comment](#) consists of <!-- + *text* + -->, where *text* does not start with > or -->, does not end with --, and does not contain --. (See the [HTML5 spec](#).)

A [processing instruction](#) consists of the string <?, a string of characters not including the string ?>, and the string ?>.

A [declaration](#) consists of the string <!, a name consisting of one or more uppercase ASCII letters, [whitespace](#), a string of characters not including the character >, and the character >.

A [CDATA section](#) consists of the string <![CDATA[, a string of characters not including the string ]]>, and the string ]]>.

An [HTML tag](#) consists of an [open tag](#), a [closing tag](#), an [HTML comment](#), a [processing instruction](#), a [declaration](#), or a [CDATA section](#).

Here are some simple open tags:

[Example 609](#)

<a><bab><c2c>

<p><a><bab><c2c></p>

Empty elements:

[Example 610](#)

`<a/><b2/>`

`<p><a/><b2/></p>`

[Whitespace](#) is allowed:

[Example 611](#)

`<a •><b2  
data="foo" >`

`<p><a •><b2  
data="foo" ></p>`

With attributes:

[Example 612](#)

`<a ffoo="bar" bam = : baz <em>"</em>  
_boolean zoop:33=zoop:33 >`

`<p><a ffoo="bar" bam = : baz <em>"</em>  
_boolean zoop:33=zoop:33 ></p>`

Custom tag names can be used:

[Example 613](#)

`Foo <responsive-image src="foo.jpg" >`

`<p>Foo <responsive-image src="foo.jpg" >  
</p>`

Illegal tag names, not parsed as HTML:

[Example 614](#)

`<33> <_>`

`<p>&lt;33&gt; &lt;_&gt;</p>`

Illegal attribute names:

[Example 615](#)

`<a h*#ref="hi">`

`<p>&lt;a h*#ref=&quot;hi&quot;&gt;</p>`

Illegal attribute values:

[Example 616](#)

`<a href="hi'> <a href=hi'>`

`<p>&lt;a href=&quot;hi'&gt; &lt;a •  
href=hi'&gt;</p>`

Illegal [whitespace](#):

[Example 617](#)

`< a><  
foo><bar/ >  
<foo bar=baz  
bim!bop >`

`<p>&lt; a&gt;&lt;  
foo&gt;&lt;bar/ &gt;  
&lt;foo bar=baz  
bim!bop >&gt;</p>`

Missing [whitespace](#):

[Example 618](#)

```

```

```
<p></p>
```

Closing tags:

[Example 619](#)

```
</foo >
```

```
<p></foo ></p>
```

Illegal attributes in closing tag:

[Example 620](#)

```
</a href="foo">
```

```
<p></a href="foo"></p>
```

Comments:

[Example 621](#)

```
foo <!-- this is a
comment - with hyphen -->
```

```
<p>foo <!-- this is a
comment - with hyphen --></p>
```

[Example 622](#)

```
foo <!-- not a comment -- two hyphens -->
```

```
<p>foo <!-- not a comment -- two hyphens -
--></p>
```

Not comments:

[Example 623](#)

```
foo <!--> foo -->
```

```
<p>foo <!--> foo --></p>
```

```
foo <!-- foo-->
```

```
<p>foo <!-- foo--></p>
```

Processing instructions:

[Example 624](#)

```
foo <?php echo $a; ?>
```

```
<p>foo <?php echo $a; ?></p>
```

Declarations:

[Example 625](#)

```
foo <!ELEMENT br EMPTY>
```

```
<p>foo <!ELEMENT br EMPTY></p>
```

CDATA sections:

[Example 626](#)

foo <![CDATA[>&<]]>

<p>foo <![CDATA[>&<]]></p>

Entity and numeric character references are preserved in HTML attributes:

#### [Example 627](#)

foo <a href="&ouml;">

<p>foo <a href="&ouml;"></p>

Backslash escapes do not work in HTML attributes:

#### [Example 628](#)

foo <a href="\\*">

<p>foo <a href="\\*"></p>

#### [Example 629](#)

<a href="\'">

<p>&lt;a href=&quot;&quot;&quot;&gt;</p>

## 6.9 Hard line breaks

A line break (not in a code span or HTML tag) that is preceded by two or more spaces and does not occur at the end of a block is parsed as a [hard line break](#) (rendered in HTML as a `<br />` tag):

#### [Example 630](#)

foo ••  
baz

<p>foo<br />  
baz</p>

For a more visible alternative, a backslash before the [line ending](#) may be used instead of two spaces:

#### [Example 631](#)

foo\  
baz

<p>foo<br />  
baz</p>

More than two spaces can be used:

#### [Example 632](#)

foo •••••  
baz

<p>foo<br />  
baz</p>

Leading spaces at the beginning of the next line are ignored:

#### [Example 633](#)

foo ••  
••••bar

<p>foo<br />  
bar</p>

#### [Example 634](#)

```
foo\
....bar
```

```
<p>foo

bar</p>
```

Line breaks can occur inside emphasis, links, and other constructs that allow inline content:

#### [Example 635](#)

```
*foo ••
bar*
```

```
<p>foo

bar</p>
```

#### [Example 636](#)

```
*foo\
bar*
```

```
<p>foo

bar</p>
```

Line breaks do not occur inside code spans

#### [Example 637](#)

```
`code •
span`
```

```
<p><code>code •span</code></p>
```

#### [Example 638](#)

```
`code\
span`
```

```
<p><code>code\ span</code></p>
```

or HTML tags:

#### [Example 639](#)

```
<a href="foo ••
bar">
```

```
<p><a href="foo ••
bar"></p>
```

#### [Example 640](#)

```
<a href="foo\
bar">
```

```
<p><a href="foo\
bar"></p>
```

Hard line breaks are for separating inline content within a block. Neither syntax for hard line breaks works at the end of a paragraph or other block element:

#### [Example 641](#)

```
foo\

```

```
<p>foo</p>
```

#### [Example 642](#)

```
foo ••

```

```
<p>foo</p>
```

#### [Example 643](#)

```
foo\

```

```
<h3>foo</h3>
```

#### [Example 644](#)



### foo ••

<h3>foo</h3>

## 6.10 Soft line breaks

A regular line break (not in a code span or HTML tag) that is not preceded by two or more spaces or a backslash is parsed as a [softbreak](#). (A softbreak may be rendered in HTML either as a [line ending](#) or as a space. The result will be the same in browsers. In the examples here, a [line ending](#) will be used.)

### [Example 645](#)

foo  
baz

<p>foo  
baz</p>

Spaces at the end of the line and beginning of the next line are removed:

### [Example 646](#)

foo •  
baz

<p>foo  
baz</p>

A conforming parser may render a soft line break in HTML either as a line break or as a space.

A renderer may also provide an option to render soft line breaks as hard line breaks.

## 6.11 Textual content

Any characters not given an interpretation by the above rules will be parsed as plain textual content.

### [Example 647](#)

hello \$. ;' there

<p>hello \$. ;' there</p>

### [Example 648](#)

Foo •x ρ ñ v

<p>Foo •x ρ ñ v </p>

Internal spaces are preserved verbatim:

### [Example 649](#)

Multiple ••••spaces

<p>Multiple ••••spaces</p>

---

## Appendix: A parsing strategy

In this appendix we describe some features of the parsing strategy used in the CommonMark reference implementations.

### Overview

Parsing has two phases:

1. In the first phase, lines of input are consumed and the block structure of the document—its division into paragraphs, block quotes, list items, and so on—is constructed. Text is assigned to these blocks but not parsed. Link reference definitions are parsed and a map of links is constructed.
2. In the second phase, the raw text contents of paragraphs and headings are parsed into sequences of Markdown inline elements (strings, code spans, links, emphasis, and so on), using the map of link references constructed in phase 1.

At each point in processing, the document is represented as a tree of **blocks**. The root of the tree is a `document` block. The `document` may have any number of other blocks as **children**. These children may, in turn, have other blocks as children. The last child of a block is normally considered **open**, meaning that subsequent lines of input can alter its contents. (Blocks that are not open are **closed**.) Here, for example, is a possible document tree, with the open blocks marked by arrows:

```
-> document
 -> block_quote
 paragraph
 "Lorem ipsum dolor\nsit amet."
 -> list (type=bullet tight=true bullet_char=--)
 list_item
 paragraph
 "Qui *quodsi iracundia*"
 -> list_item
 -> paragraph
 "aliquando id"
```

## Phase 1: block structure

Each line that is processed has an effect on this tree. The line is analyzed and, depending on its contents, the document may be altered in one or more of the following ways:

1. One or more open blocks may be closed.
2. One or more new blocks may be created as children of the last open block.
3. Text may be added to the last (deepest) open block remaining on the tree.

Once a line has been incorporated into the tree in this way, it can be discarded, so input can be read in a stream.

For each line, we follow this procedure:

1. First we iterate through the open blocks, starting with the root `document`, and descending through last children down to the last open block. Each block imposes a condition that the line must satisfy if the block is to remain open. For example, a block quote requires a `>` character. A paragraph requires a non-blank line. In this phase we may match all or just some of the open blocks. But we cannot close unmatched blocks yet, because we may have a [lazy continuation line](#).
2. Next, after consuming the continuation markers for existing blocks, we look for new block starts (e.g. `>` for a block quote). If we encounter a new block start, we close any blocks unmatched in step 1 before creating the new block as a child of the last matched block.
3. Finally, we look at the remainder of the line (after block markers like `>`, list markers, and indentation have been consumed). This is text that can be incorporated into the last open block (a paragraph, code block, heading, or raw HTML).

Setext headings are formed when we see a line of a paragraph that is a [setext heading underline](#).

Reference link definitions are detected when a paragraph is closed; the accumulated text lines are parsed to see if they begin with one or more reference link definitions. Any remainder becomes a normal paragraph.

We can see how this works by considering how the tree above is generated by four lines of Markdown:

```
> Lorem ipsum dolor
sit amet.
> - Qui *quodsi iracundia*
> - aliquando id
```

At the outset, our document model is just

```
-> document
```

The first line of our text,

```
> Lorem ipsum dolor
```

causes a `block_quote` block to be created as a child of our open `document` block, and a `paragraph` block as a child of the `block_quote`. Then the text is added to the last open block, the `paragraph`:

```
-> document
 -> block_quote
 -> paragraph
 "Lorem ipsum dolor"
```

The next line,

```
sit amet.
```

is a “lazy continuation” of the open `paragraph`, so it gets added to the `paragraph`’s text:

```
-> document
 -> block_quote
 -> paragraph
 "Lorem ipsum dolor\nsit amet."
```

The third line,

```
> - Qui *quodsi iracundia*
```

causes the `paragraph` block to be closed, and a new `list` block opened as a child of the `block_quote`. A `list_item` is also added as a child of the `list`, and a `paragraph` as a child of the `list_item`. The text is then added to the new `paragraph`:

```
-> document
 -> block_quote
 paragraph
 "Lorem ipsum dolor\nsit amet."
 -> list (type=bullet tight=true bullet_char=-)
 -> list_item
 -> paragraph
 "Qui *quodsi iracundia*"
```

The fourth line,

```
> - aliquando id
```

causes the `list_item` (and its child the `paragraph`) to be closed, and a new `list_item` opened up as child of the `list`. A `paragraph` is added as a child of the new `list_item`, to contain the text. We thus obtain the final tree:

```
-> document
 -> block_quote
 paragraph
 "Lorem ipsum dolor\nsit amet."
 -> list (type=bullet tight=true bullet_char=--)
 list_item
 paragraph
 "Qui *quodsi iracundia*"
 -> list_item
 -> paragraph
 "aliquando id"
```

## Phase 2: inline structure

Once all of the input has been parsed, all open blocks are closed.

We then “walk the tree,” visiting every node, and parse raw string contents of paragraphs and headings as inlines. At this point we have seen all the link reference definitions, so we can resolve reference links as we go.

```
document
 block_quote
 paragraph
 str "Lorem ipsum dolor"
 softbreak
 str "sit amet."
 list (type=bullet tight=true bullet_char=--)
 list_item
 paragraph
 str "Qui "
 emph
 str "quodsi iracundia"
 list_item
 paragraph
 str "aliquando id"
```

Notice how the [line ending](#) in the first paragraph has been parsed as a `softbreak`, and the asterisks in the first list item have become an `emph`.

## An algorithm for parsing nested emphasis and links

By far the trickiest part of inline parsing is handling emphasis, strong emphasis, links, and images. This is done using the following algorithm.

When we’re parsing inlines and we hit either

- a run of `*` or `_` characters, or
- a `[` or `! [`

we insert a text node with these symbols as its literal content, and we add a pointer to this text node to the [delimiter stack](#).

The [delimiter stack](#) is a doubly linked list. Each element contains a pointer to a text node, plus information about

- the type of delimiter ([, ![, \*, \_)
- the number of delimiters,
- whether the delimiter is “active” (all are active to start), and
- whether the delimiter is a potential opener, a potential closer, or both (which depends on what sort of characters precede and follow the delimiters).

When we hit a ] character, we call the *look for link or image* procedure (see below).

When we hit the end of the input, we call the *process emphasis* procedure (see below), with `stack_bottom = NULL`.

### ***look for link or image***

Starting at the top of the delimiter stack, we look backwards through the stack for an opening [ or ![ delimiter.

- If we don’t find one, we return a literal text node ].
- If we do find one, but it’s not *active*, we remove the inactive delimiter from the stack, and return a literal text node ].
- If we find one and it’s active, then we parse ahead to see if we have an inline link/image, reference link/image, compact reference link/image, or shortcut reference link/image.
  - If we don’t, then we remove the opening delimiter from the delimiter stack and return a literal text node ].
  - If we do, then
    - We return a link or image node whose children are the inlines after the text node pointed to by the opening delimiter.
    - We run *process emphasis* on these inlines, with the [ opener as `stack_bottom`.
    - We remove the opening delimiter.
    - If we have a link (and not an image), we also set all [ delimiters before the opening delimiter to *inactive*. (This will prevent us from getting links within links.)

### ***process emphasis***

Parameter `stack_bottom` sets a lower bound to how far we descend in the [delimiter stack](#). If it is NULL, we can go all the way to the bottom. Otherwise, we stop before visiting `stack_bottom`.

Let `current_position` point to the element on the [delimiter stack](#) just above `stack_bottom` (or the first element if `stack_bottom` is NULL).

We keep track of the `openers_bottom` for each delimiter type (\*, \_) and each length of the closing delimiter run (modulo 3). Initialize this to `stack_bottom`.

Then we repeat the following until we run out of potential closers:

- Move `current_position` forward in the delimiter stack (if needed) until we find the first potential closer with delimiter \* or \_. (This will be the potential closer closest to the beginning of the input – the first one in parse order.)

- Now, look back in the stack (staying above `stack_bottom` and the `openers_bottom` for this delimiter type) for the first matching potential opener (“matching” means same delimiter).
- If one is found:
  - Figure out whether we have emphasis or strong emphasis: if both closer and opener spans have length  $\geq 2$ , we have strong, otherwise regular.
  - Insert an `emph` or `strong emph` node accordingly, after the text node corresponding to the opener.
  - Remove any delimiters between the opener and closer from the delimiter stack.
  - Remove 1 (for regular `emph`) or 2 (for strong `emph`) delimiters from the opening and closing text nodes. If they become empty as a result, remove them and remove the corresponding element of the delimiter stack. If the closing node is removed, reset `current_position` to the next element in the stack.
- If none is found:
  - Set `openers_bottom` to the element before `current_position`. (We know that there are no openers for this kind of closer up to and including this point, so this puts a lower bound on future searches.)
  - If the closer at `current_position` is not a potential opener, remove it from the delimiter stack (since we know it can't be a closer either).
  - Advance `current_position` to the next element in the stack.

After we're done, we remove all delimiters above `stack_bottom` from the delimiter stack.