CommonMark Spec

Version 0.31.2 (2024-01-28)
John MacFarlane

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. (https://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.

Note that not every feature of the HTML samples is mandated by the spec. For example, the spec says what counts as a link destination, but it doesn’t mandate that non-ASCII characters in the URL be percent-encoded. To use the automatic tests, implementers will need to provide a renderer that conforms to the expectations of the spec examples (percent-encoding non-ASCII characters in URLs). But a conforming implementation can use a different renderer and may choose not to percent-encode non-ASCII characters in URLs.

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 line feed (`U+000A`) or carriage return (`U+000D`), followed by a line ending or by the end of file.

A **line ending** is a line feed (`U+000A`), a carriage return (`U+000D`) not followed by a line feed, or a carriage return and a following line feed.

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 **Unicode whitespace character** is a character in the Unicode `Zs` general category, or a tab (`U+0009`), line feed (`U+000A`), form feed (`U+000C`), or carriage return (`U+000D`).

Unicode whitespace is a sequence of one or more Unicode whitespace characters.

A **tab** is `U+0009`.

A **space** is `U+0020`.

An **ASCII control character** is a character between `U+0000–1F` (both including) or `U+007F`.

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

A **Unicode punctuation character** is a character in the Unicode `P` (puncuation) or `S` (symbol) general categories.

2.2 Tabs

Tabs in lines are not expanded to spaces. However, in contexts where spaces help 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

```
→foo→baz→→bim
```

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

Example 2

```
··→foo→baz→→bim
```

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

Example 3

```
····a→a
····ὐ→a
```

```
<pre><code>a→a
ὐ→a
</code></pre>
```

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

```
··-·foo

→bar
```

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

Example 5

```
-·foo

→→bar
```

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

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
```

```
<ul>
<li>
<pre><code>··foo
</code></pre>
</li>
</ul>
```

Example 8

```
····foo
→bar
```

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

Example 9

```
·-·foo
···-·bar
→·-·baz
```

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

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`).

2.4 Backslash escapes

Any ASCII punctuation character may be backslash-escaped:

Example 12

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

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

Backslashes before other characters are treated as literal backslashes:

Example 13

```
\→\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 14

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

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

Example 15

```
\\*emphasis*
```

```
<p>\<em>emphasis</em></p>
```

A backslash at the end of the line is a hard line break:

Example 16

```
foo\
bar
```

```
<p>foo<br·/>
bar</p>
```

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

Example 17

```
``·\[\`·``
```

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

Example 18

```
····\[\]
```

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

Example 19

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

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

Example 20

```
<https://example.com?find=\*>
```

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

Example 21

```
<a·href="/bar\/)">
```

```
<a·href="/bar\/)">
```

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

Example 22

```
[foo](/bar\*·"ti\*tle")
```

```
<p><a·href="/bar*"·
title="ti*tle">foo</a></p>
```

Example 23

```
[foo]

[foo]:·/bar\*·"ti\*tle"
```

```
<p><a·href="/bar*"·
title="ti*tle">foo</a></p>
```

Example 24

~~~
```·foo\+bar
foo
```
~~~

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

2.5 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 `&#42;` can be used in place of a literal `*` character, `&#42;` 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/entities.json is used as an authoritative source for the valid entity references and their corresponding code points.

Example 25

```
&nbsp;·&amp;·&copy;·&AElig;·&Dcaron;     <p> ·&amp;·©·Æ·Ď
&frac34;·&HilbertSpace;·                 ¾·ℋ·ⅆ
&DifferentialD;                          ∲·≧̸</p>
&ClockwiseContourIntegral;·&ngE;
```

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 26

```
&#35;·&#1234;·&#992;·&#0;                <p>#·Ӓ·Ϡ·�</p>
```

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 27

```
&#X22;·&#XD06;·&#xcab;                   <p>&quot;·ആ·ಫ</p>
```

Here are some nonentities:

Example 28

```
&nbsp·&x;·&#;·&#x;                       <p>&amp;nbsp·&amp;x;·&amp;#;·&amp;#x;
&#87654321;                              &amp;#87654321;
&#abcdef0;                               &amp;#abcdef0;
&ThisIsNotDefined;·&hi?;                 &amp;ThisIsNotDefined;·&amp;hi?;</p>
```

Although HTML5 does accept some entity references without a trailing semicolon (such as `&copy`), these are not recognized here, because it makes the grammar too ambiguous:

Example 29

```
&copy                                    <p>&amp;copy</p>
```

Strings that are not on the list of HTML5 named entities are not recognized as entity references either:

Example 30

```
&MadeUpEntity;                           <p>&amp;MadeUpEntity;</p>
```

Entity and numeric character references are recognized in any context besides code spans or code blocks, including URLs, link titles, and fenced code block info strings:

Example 31

```
<a·href="&ouml;&ouml;.html">
```

```
<a·href="&ouml;&ouml;.html">
```

Example 32

```
[foo](/f&ouml;&ouml;·"f&ouml;&ouml;")
```

```
<p><a·href="/f%C3%B6%C3%B6"·
title="föö">foo</a></p>
```

Example 33

```
[foo]

[foo]:·/f&ouml;&ouml;·"f&ouml;&ouml;"
```

```
<p><a·href="/f%C3%B6%C3%B6"·
title="föö">foo</a></p>
```

Example 34

~~~
```·f&ouml;&ouml;
foo
```
~~~

```
<pre><code·class="language-föö">foo
</code></pre>
```

Entity and numeric character references are treated as literal text in code spans and code blocks:

Example 35

```
`f&ouml;&ouml;`
```

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

Example 36

```
····f&ouml;f&ouml;
```

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

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

Example 37

```
&#42;foo&#42;
*foo*
```

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

Example 38

```
&#42;·foo

*·foo
```

```
<p>*·foo</p>
<ul>
<li>foo</li>
</ul>
```

Example 39

```
foo&#10;&#10;bar                <p>foo

                                  bar</p>
```

Example 40

```
&#9;foo                           <p>→foo</p>
```

Example 41

```
[a](url·&quot;tit&quot;)         <p>[a](url·&quot;tit&quot;)</p>
```

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 42

```
-·`one                            <ul>
-·two`                            <li>`one</li>
                                  <li>two`</li>
                                  </ul>
```

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 optionally up to three 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 43

```
***
---
___
```

```
<hr·/>
<hr·/>
<hr·/>
```

Wrong characters:

Example 44

```
+++
```

```
<p>+++</p>
```

Example 45

```
===
```

```
<p>===</p>
```

Not enough characters:

Example 46

```
--
**
__
```

```
<p>--
**
__</p>
```

Up to three spaces of indentation are allowed:

Example 47

```
·***
··***
···***
```

```
<hr·/>
<hr·/>
<hr·/>
```

Four spaces of indentation is too many:

Example 48

```
····***
```

```
<pre><code>***
</code></pre>
```

Example 49

```
Foo
····***
```

```
<p>Foo
***</p>
```

More than three characters may be used:

[Example 50](#)

```
_____________________________________                <hr·/>
```

Spaces and tabs are allowed between the characters:

[Example 51](#)

```
·-·-·-                                              <hr·/>
```

[Example 52](#)

```
·**··*·**·*·**·*·**                                 <hr·/>
```

[Example 53](#)

```
-·····-······-······-                               <hr·/>
```

Spaces and tabs are allowed at the end:

[Example 54](#)

```
-·-·-·····                                          <hr·/>
```

However, no other characters may occur in the line:

[Example 55](#)

```
_·_·_·_·a                                           <p>_·_·_·_·a</p>
                                                    <p>a------</p>
a------                                             <p>---a---</p>

---a---
```

It is required that all of the characters other than spaces or tabs be the same. So, this is not a thematic break:

[Example 56](#)

```
·*-*                                                <p><em>-</em></p>
```

Thematic breaks do not need blank lines before or after:

[Example 57](#)

```
-·foo
***
-·bar
```

```
<ul>
<li>foo</li>
</ul>
<hr·/>
<ul>
<li>bar</li>
</ul>
```

Thematic breaks can interrupt a paragraph:

Example 58

```
Foo
***
bar
```

```
<p>Foo</p>
<hr·/>
<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 59

```
Foo
---
bar
```

```
<h2>Foo</h2>
<p>bar</p>
```

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

Example 60

```
*·Foo
*·*·*
*·Bar
```

```
<ul>
<li>Foo</li>
</ul>
<hr·/>
<ul>
<li>Bar</li>
</ul>
```

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

Example 61

```
-·Foo
-·*·*·*
```

```
<ul>
<li>Foo</li>
<li>
<hr·/>
</li>
</ul>
```

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 spaces or tabs, or by the end of line. The optional closing sequence of #s must be preceded by spaces or tabs and may be followed by spaces or tabs only. The opening # character may be preceded by up to three spaces of indentation. The raw contents of the heading are stripped of leading and trailing space or tabs before being parsed as inline content. The heading level is equal to the number of # characters in the opening sequence.

Simple headings:

Example 62

```
#·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 63

```
#######·foo                             <p>#######·foo</p>
```

At least one space or tab 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 64

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

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

Example 65

```
\##·foo                                 <p>##·foo</p>
```

Contents are parsed as inlines:

Example 66

```
#·foo·*bar*·\*baz\*                      <h1>foo·<em>bar</em>·*baz*</h1>
```

Leading and trailing spaces or tabs are ignored in parsing inline content:

Example 67

```
#··················foo················   <h1>foo</h1>
·····
```

Up to three spaces of indentation are allowed:

Example 68

```
·###·foo                                 <h3>foo</h3>
··##·foo                                 <h2>foo</h2>
···#·foo                                 <h1>foo</h1>
```

Four spaces of indentation is too many:

Example 69

```
····#·foo                                <pre><code>#·foo
                                         </code></pre>
```

Example 70

```
foo                                      <p>foo
····#·bar                                #·bar</p>
```

A closing sequence of # characters is optional:

Example 71

```
##·foo·##                                <h2>foo</h2>
··###···bar····###                       <h3>bar</h3>
```

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

Example 72

```
#·foo·                                   <h1>foo</h1>
##################################       <h5>foo</h5>
#####·foo·##
```

Spaces or tabs are allowed after the closing sequence:

Example 73

```
###·foo·###·····                        <h3>foo</h3>
```

A sequence of # characters with anything but spaces or tabs following it is not a closing sequence, but counts as part of the contents of the heading:

Example 74

```
###·foo·###·b                           <h3>foo·###·b</h3>
```

The closing sequence must be preceded by a space or tab:

Example 75

```
#·foo#                                  <h1>foo#</h1>
```

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

Example 76

```
###·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 77

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

Example 78

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

ATX headings can be empty:

Example 79

```
##·                                     <h2></h2>
#                                       <h1></h1>
###·###                                 <h3></h3>
```

4.3 Setext headings

A **setext heading** consists of one or more lines of text, not interrupted by a blank line, of which the first line does not have more than 3 spaces of 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 of indentation and any number of trailing spaces or tabs.

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 80

```
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 81

```
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 spaces or tabs.

Example 82

```
··Foo·*bar
baz*→
====
```

```
<h1>Foo·<em>bar
baz</em></h1>
```

The underlining can be any length:

Example 83

```
Foo
-------------------------

Foo
=
```

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

The heading content can be preceded by up to three spaces of indentation, and need not line up with the underlining:

Example 84

```
···Foo
---

··Foo
-----

··Foo
··===
```

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

Four spaces of indentation is too many:

Example 85

```
····Foo
····---

····Foo
---
```

```
<pre><code>Foo
---

Foo
</code></pre>
<hr·/>
```

The setext heading underline can be preceded by up to three spaces of indentation, and may have trailing spaces or tabs:

Example 86

```
Foo
···----······
```

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

Four spaces of indentation is too many:

Example 87

```
Foo
····---
```

```
<p>Foo
---</p>
```

The setext heading underline cannot contain internal spaces or tabs:

Example 88

```
Foo
=·=

Foo
---·-
```

```
<p>Foo
=·=</p>
<p>Foo</p>
<hr·/>
```

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

Example 89

```
Foo··
-----
```

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

Nor does a backslash at the end:

Example 90

```
Foo\
----
```

```
<h2>Foo\</h2>
```

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

Example 91

```
`Foo
----
`

<a·title="a·lot
---
of·dashes"/>
```

```
<h2>`Foo</h2>
<p>`</p>
<h2>&lt;a·title=&quot;a·lot</h2>
<p>of·dashes&quot;/&gt;</p>
```

The setext heading underline cannot be a lazy continuation line in a list item or block quote:

Example 92

```
>·Foo
---
```

```
<blockquote>
<p>Foo</p>
</blockquote>
<hr·/>
```

Example 93

```
>·foo
bar
===
```

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

Example 94

```
-·Foo                                   <ul>
---                                     <li>Foo</li>
                                        </ul>
                                        <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 95

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

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

Example 96

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

Setext headings cannot be empty:

Example 97

```
                                        <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 98

```
---                                     <hr·/>
---                                     <hr·/>
```

Example 99

```
-·foo                                   <ul>
-----                                   <li>foo</li>
                                        </ul>
                                        <hr·/>
```

Example 100

```
····foo
---
```

```
<pre><code>foo
</code></pre>
<hr·/>
```

Example 101

```
>·foo
-----
```

```
<blockquote>
<p>foo</p>
</blockquote>
<hr·/>
```

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

Example 102

```
\>·foo
------
```

```
<h2>&gt;·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 103

```
Foo

bar
---
baz
```

```
<p>Foo</p>
<h2>bar</h2>
<p>baz</p>
```

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

Example 104

```
Foo                                     <p>Foo
bar                                     bar</p>
                                        <hr·/>
---                                     <p>baz</p>

baz
```

or use a thematic break that cannot count as a setext heading underline, such as

Example 105

```
Foo                                     <p>Foo
bar                                     bar</p>
*·*·*                                   <hr·/>
baz                                     <p>baz</p>
```

Authors who want interpretation 3 can use backslash escapes:

Example 106

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

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 preceded by four or more spaces of indentation. 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 107

```
····a·simple                            <pre><code>a·simple
······indented·code·block               ··indented·code·block
                                        </code></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 108

```
··-·foo

····bar
```

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

Example 109

```
1.··foo

····-·bar
```

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

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

Example 110

```
····<a/>
····*hi*

····-·one
```

```
<pre><code>&lt;a/&gt;
*hi*

-·one
</code></pre>
```

Here we have three chunks separated by blank lines:

Example 111

```
····chunk1

····chunk2
··
·
·
····chunk3
```

```
<pre><code>chunk1

chunk2



chunk3
</code></pre>
```

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

Example 112

```
····chunk1
······
······chunk2
```

```
<pre><code>chunk1
··
··chunk2
</code></pre>
```

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

Example 113

```
Foo
····bar
```

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

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

Example 114

```
····foo
bar
```

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

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

Example 115

```
#·Heading
····foo
Heading
------
····foo
----
```

```
<h1>Heading</h1>
<pre><code>foo
</code></pre>
<h2>Heading</h2>
<pre><code>foo
</code></pre>
<hr·/>
```

The first line can be preceded by more than four spaces of indentation:

Example 116

```
········foo
····bar
```

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

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

Example 117

```

····
····foo
····
```

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

Trailing spaces or tabs are included in the code block’s content:

Example 118

```
····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, preceded by up to three spaces of indentation.

The line with the opening code fence may optionally contain some text following the code fence; this is trimmed of leading and trailing spaces or tabs 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 preceded by N spaces of indentation, 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 N spaces or less, all of the indentation is removed.)

The closing code fence may be preceded by up to three spaces of indentation, and may be followed only by spaces or tabs, 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 downside 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 119

~~~
```
<
·>
```
~~~

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

With tildes:

Example 120

```
~~~
<
·>
~~~
```

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

Fewer than three backticks is not enough:

Example 121

```````
``
foo
``
```````

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

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

Example 122

```````
```
aaa
~~~
```
```````

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

Example 123

```````
~~~
aaa
```
~~~
```````

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

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

Example 124

~~~
````
aaa
```
``````
~~~

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

Example 125

```````
~~~~
aaa
~~~
~~~~
```````

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

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

Example 126

~~~
```
~~~

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

Example 127
```````

~~~
`````

```
aaa
~~~

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

Example 128

````
>·```
>·aaa

bbb
````

```
<blockquote>
<pre><code>aaa
</code></pre>
</blockquote>
<p>bbb</p>
```

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

Example 129

~~~
```

··
```
~~~

```
<pre><code>
··
</code></pre>
```

A code block can be empty:

Example 130

~~~
```
```
~~~

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

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

Example 131

~~~
·```
·aaa
aaa
```
~~~

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

Example 132

````
··```
aaa
··aaa
aaa
··```
````

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

Example 133

```
···```
···aaa
····aaa
··aaa
···```
```

```
<pre><code>aaa
·aaa
aaa
</code></pre>
```

Four spaces of indentation is too many:

Example 134

```
····```
····aaa
····```
```

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

Closing fences may be preceded by up to three spaces of indentation, and their indentation need not match that of the opening fence:

Example 135

```
```
aaa
··```
```

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

Example 136

```
···```
aaa
··```
```

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

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

Example 137

```
```
aaa
····```
```

```
<pre><code>aaa
····```
</code></pre>
```

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

Example 138

```
```·```
aaa
```

```
<p><code>·</code>
aaa</p>
```

Example 139

```
~~~~~~
aaa
~~~·~~
```

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

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

Example 140

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

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

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

Example 141

```
foo
---
~~~
bar
~~~
#·baz
```

```
<h2>foo</h2>
<pre><code>bar
</code></pre>
<h1>baz</h1>
```

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 142

~~~
```ruby
def·foo(x)
··return·3
end
```
~~~

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

Example 143

```
~~~~····ruby·startline=3·$%@#$
def·foo(x)
··return·3
end
~~~~~~~
```

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

Example 144

~~~
````;
````
~~~

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

Info strings for backtick code blocks cannot contain backticks:

Example 145

~~~
```·aa·```
foo
~~~

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

Info strings for tilde code blocks can contain backticks and tildes:

Example 146

```
~~~·aa·```·~~~
foo
~~~
```

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

Closing code fences cannot have info strings:

Example 147

~~~
```
```·aaa
```
~~~

````
<pre><code>```·aaa
</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 optional spaces of 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 `<pre`, `<script`, `<style`, or `<textarea` (case-insensitive), followed by a space, a tab, the string `>`, or the end of the line.
   End condition: line contains an end tag `</pre>`, `</script>`, `</style>`, or `</textarea>` (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 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 with 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, search, section, summary, table, tbody, td, tfoot, th, thead, title, tr, track, ul, followed by a space, a tab, 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 pre, script, style, or textarea) or a complete closing tag, followed by zero or more spaces and tabs, followed by 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 an 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 148

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

_world_.
</pre>
</td></tr></table>
```

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

In this case, the HTML block is terminated by the blank line — 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 149

```
<table>
··<tr>
····<td>
··········hi
····</td>
··</tr>
</table>

okay.
```

```
<table>
··<tr>
····<td>
··········hi
····</td>
··</tr>
</table>
<p>okay.</p>
```

Example 150

```
·<div>
··*hello*
·········<foo><a>
```

```
·<div>
··*hello*
·········<foo><a>
```

A block can also start with a closing tag:

Example 151

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

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

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

Example 152

```
<DIV·CLASS="foo">

*Markdown*

</DIV>
```

```
<DIV·CLASS="foo">
<p><em>Markdown</em></p>
</DIV>
```

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

Example 153

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

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

Example 154

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

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

An open tag need not be closed:

Example 155

```
<div>
*foo*

*bar*
```

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

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

Example 156

```
<div·id="foo"
*hi*
```

```
<div·id="foo"
*hi*
```

Example 157

```
<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 158

```
<div·*???-&&&-<---
*foo*
```

```
<div·*???-&&&-<---
*foo*
```

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

Example 159

```
<div><a·href="bar">*foo*</a></div>
```

```
<div><a·href="bar">*foo*</a></div>
```

Example 160

```
<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 161

~~~
<div></div>
```·c
int·x·=·33;
~~~
```

```
<div></div>
```·c
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 162

```
<a·href="foo">
*bar*
</a>
```

```
<a·href="foo">
*bar*
</a>
```

In type 7 blocks, the tag name can be anything:

Example 163

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

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

Example 164

```
<i·class="foo">
*bar*
</i>
```

```
<i·class="foo">
*bar*
</i>
```

Example 165

```
</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 166

```
<del>
*foo*
</del>
```

```
<del>
*foo*
</del>
```

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 167

```
<del>

*foo*

</del>
```

```
<del>
<p><em>foo</em></p>
</del>
```

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 168

```
<del>*foo*</del>
```

```
<p><del><em>foo</em></del></p>
```

HTML tags designed to contain literal content (`pre`, `script`, `style`, `textarea`), 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 169

```
<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 170

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

document.getElementById("demo").inner
HTML·=·"Hello·JavaScript!";
</script>
okay
```

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

document.getElementById("demo").inner
HTML·=·"Hello·JavaScript!";
</script>
<p>okay</p>
```

A textarea tag (type 1):

Example 171

```
<textarea>

*foo*

_bar_

</textarea>
```

```
<textarea>

*foo*

_bar_

</textarea>
```

A style tag (type 1):

Example 172

```
<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 173

```
<style
··type="text/css">

foo
```

```
<style
··type="text/css">

foo
```

Example 174

```
>·<div>
>·foo

bar
```

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

Example 175

```
-·<div>
-·foo
```

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

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

Example 176

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

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

Example 177

```
<!--·foo·-->*bar*
*baz*
```

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

Note that anything on the last line after the end tag will be included in the HTML block:

Example 178

```
<script>
foo
</script>1.·*bar*
```

```
<script>
foo
</script>1.·*bar*
```

A comment (type 2):

Example 179

```
<!--·Foo

bar
···baz·-->
okay
```

```
<!--·Foo

bar
···baz·-->
<p>okay</p>
```

A processing instruction (type 3):

Example 180

```
<?php

··echo·'>';

?>
okay
```

```
<?php

··echo·'>';

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

A declaration (type 4):

Example 181

```
<!DOCTYPE·html>
```

```
<!DOCTYPE·html>
```

CDATA (type 5):

Example 182

```
<![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 preceded by up to three spaces of indentation, but not four:

Example 183

```
··<!--·foo·-->

····<!--·foo·-->
```

```
··<!--·foo·-->
<pre><code>&lt;!--·foo·--&gt;
</code></pre>
```

Example 184

```
··<div>

····<div>
```

```
··<div>
<pre><code>&lt;div&gt;
</code></pre>
```

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

Example 185

```
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 186

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

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

HTML blocks of type 7 cannot interrupt a paragraph:

Example 187

```
Foo
<a·href="bar">
baz
```

```
<p>Foo
<a·href="bar">
baz</p>
```

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 spaces or tabs.

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 188

```
<div>

*Emphasized*·text.

</div>
```

```
<div>
<p><em>Emphasized</em>·text.</p>
</div>
```

Example 189

```
<div>
*Emphasized*·text.
</div>
```

```
<div>
*Emphasized*·text.
</div>
```

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 190

```
<table>

<tr>

<td>
Hi
</td>

</tr>

</table>
```

```
<table>
<tr>
<td>
Hi
</td>
</tr>
</table>
```

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 191

```
<table>

··<tr>

····<td>
······Hi
····</td>

··</tr>

</table>
```

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

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, optionally preceded by up to three spaces of indentation, followed by a colon (`:`), optional spaces or tabs (including up to one line ending), a link destination, optional spaces or tabs (including up to one line ending), and an optional link title, which if it is present must be separated from the link destination by spaces or tabs. No further character may occur.

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 192

```
[foo]:·/url·"title"

[foo]
```

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

Example 193

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

[foo]
```

```
<p><a·href="/url"·title="the·
title">foo</a></p>
```

Example 194

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

[Foo*bar\]]
```

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

Example 195

```
[Foo·bar]:
<my·url>
'title'

[Foo·bar]
```

```
<p><a·href="my%20url"·
title="title">Foo·bar</a></p>
```

The title may extend over multiple lines:

Example 196

```
[foo]:·/url·'
title
line1
line2
'

[foo]
```

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

However, it may not contain a blank line:

Example 197

```
[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 198

```
[foo]:
/url

[foo]
```

```
<p><a·href="/url">foo</a></p>
```

The link destination may not be omitted:

Example 199

```
[foo]:

[foo]
```

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

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

Example 200

```
[foo]:·<>

[foo]
```

```
<p><a·href="">foo</a></p>
```

The title must be separated from the link destination by spaces or tabs:

Example 201

```
[foo]:·<bar>(baz)

[foo]
```

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

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

Example 202

```
[foo]:·/url\bar\*baz·"foo\"bar\baz"

[foo]
```

```
<p><a·href="/url%5Cbar*baz"·
title="foo&quot;bar\baz">foo</a></p>
```

A link can come before its corresponding definition:

Example 203

```
[foo]

[foo]:·url
```

```
<p><a·href="url">foo</a></p>
```

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

Example 204

```
[foo]

[foo]:·first
[foo]:·second
```

```
<p><a·href="first">foo</a></p>
```

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

Example 205

```
[FOO]:·/url

[Foo]
```

```
<p><a·href="/url">Foo</a></p>
```

Example 206

```
[ΑΓΩ]:·/φου

[αγω]
```

```
<p><a·
href="/%CF%86%CE%BF%CF%85">αγω</a>
</p>
```

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 207

```
[foo]:·/url
```

Here is another one:

Example 208

```
[
foo
]:·/url
bar
```

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

This is not a link reference definition, because there are characters other than spaces or tabs after the title:

Example 209

```
[foo]:·/url·"title"·ok
```

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

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

Example 210

```
[foo]:·/url
"title"·ok
```

```
<p>&quot;title&quot;·ok</p>
```

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

Example 211

```
····[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 212

~~~
```
[foo]:·/url
```

[foo]
~~~

```
<pre><code>[foo]:·/url
</code></pre>
<p>[foo]</p>
```

A link reference definition cannot interrupt a paragraph.

Example 213

```
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 214

```
#·[Foo]
[foo]:·/url
>·bar
```

```
<h1><a·href="/url">Foo</a></h1>
<blockquote>
<p>bar</p>
</blockquote>
```

Example 215

```
[foo]:·/url
bar
===
[foo]
```

```
<h1>bar</h1>
<p><a·href="/url">foo</a></p>
```

Example 216

```
[foo]:·/url
===
[foo]
```

```
<p>===
<a·href="/url">foo</a></p>
```

Several link reference definitions can occur one after another, without intervening blank lines.

Example 217

```
[foo]:·/foo-url·"foo"
[bar]:·/bar-url
··"bar"
[baz]:·/baz-url

[foo],
[bar],
[baz]
```

```
<p><a·href="/foo-url"·
title="foo">foo</a>,
<a·href="/bar-url"·
title="bar">bar</a>,
<a·href="/baz-url">baz</a></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 218

```
[foo]

>·[foo]:·/url
```

```
<p><a·href="/url">foo</a></p>
<blockquote>
</blockquote>
```

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 spaces or tabs.

A simple example with two paragraphs:

Example 219

```
aaa

bbb
```

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

Paragraphs can contain multiple lines, but no blank lines:

Example 220

```
aaa
bbb

ccc
ddd
```

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

Multiple blank lines between paragraphs have no effect:

Example 221

```
aaa


bbb
```

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

Leading spaces or tabs are skipped:

Example 222

```
··aaa
·bbb
```

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

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

Example 223

```
aaa
·············bbb
········································
·ccc
```

```
<p>aaa
bbb
ccc</p>
```

However, the first line may be preceded by up to three spaces of indentation. Four spaces of indentation is too many:

Example 224

```
···aaa
bbb
```

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

Example 225

```
····aaa
bbb
```

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

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

Example 226

```
aaa·····
bbb·····
```

```
<p>aaa<br·/>
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 227

```
··

aaa
··

#·aaa

··
```

```
<p>aaa</p>
<h1>aaa</h1>
```

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**, optionally preceded by up to three spaces of indentation, consists of (a) the character > together with a following space of indentation, or (b) a single character > not followed by a space of indentation.

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 character other than a space or tab 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 228

```
>·#·Foo
>·bar
>·baz
```

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

The space or tab after the > characters can be omitted:

Example 229

```
>#·Foo
>bar
>·baz
```

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

The > characters can be preceded by up to three spaces of indentation:

Example 230

```
···>·#·Foo
···>·bar
·>·baz
```

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

Four spaces of indentation is too many:

Example 231

```
····>·#·Foo
····>·bar
····>·baz
```

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

The Laziness clause allows us to omit the > before paragraph continuation text:

Example 232

```
>·#·Foo
>·bar
baz
```

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

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

Example 233

```
>·bar
baz
>·foo
```

```
<blockquote>
<p>bar
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 234

```
>·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 235

```
>·-·foo
-·bar
```

```
<blockquote>
<ul>
<li>foo</li>
</ul>
</blockquote>
<ul>
<li>bar</li>
</ul>
```

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

Example 236

```
>·····foo
····bar
```

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

Example 237

```
>·```
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 238
```

```
>·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 239

```
>
```

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

Example 240

```
>
>··
>·
```

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

A block quote can have initial or final blank lines:

Example 241

```
>
>·foo
>··
```

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

A blank line always separates block quotes:

Example 242

```
>·foo

>·bar
```

```
<blockquote>
<p>foo</p>
</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 243

```
>·foo
>·bar
```

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

To get a block quote with two paragraphs, use:

Example 244

```
>·foo
>
>·bar
```

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

Block quotes can interrupt paragraphs:

Example 245

```
foo
>·bar
```

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

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

Example 246

```
>·aaa
***
>·bbb
```

```
<blockquote>
<p>aaa</p>
</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 247

```
>·bar
baz
```

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

Example 248

```
>·bar                                  <blockquote>
                                       <p>bar</p>
baz                                    </blockquote>
                                       <p>baz</p>
```

Example 249

```
>·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 250

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

Example 251

```
>>>·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 of indentation. So *five spaces* are needed after the >:

Example 252

```
>·····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 character other than a space or tab, and *M* is a list marker of width *W* followed by 1 ≤ *N* ≤ 4 spaces of indentation, 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 253

```
A·paragraph
with·two·lines.

····indented·code

>·A·block·quote.
```

```
<p>A·paragraph
with·two·lines.</p>
<pre><code>indented·code
</code></pre>
<blockquote>
<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 254

```
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>
```

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 of indentation, and there are three spaces between the list marker and the next character other than a space or tab, 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 255

```
-·one

·two
```

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

Example 256

```
-·one

··two
```

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

Example 257

```
·-····one

·····two
```

```
<ul>
<li>one</li>
</ul>
<pre><code>·two
</code></pre>
```

Example 258

```
·-····one

······two
```

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

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 character other than a space or tab after the list marker. However, that is not quite right. The spaces of indentation 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 259

```
···>·>·1.··one
>>
>>·····two
```

```
<blockquote>
<blockquote>
<ol>
<li>
<p>one</p>
<p>two</p>
</li>
</ol>
</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 260

```
>>-·one
>>
··>··>·two
```

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

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

Example 261

```
-one

2.two
```

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

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

Example 262

```
-·foo


··bar
```

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

A list item may contain any kind of block:

Example 263

```
1.··foo

····```
····bar
····```

····baz

····>·bam
```

```html
<ol>
<li>
<p>foo</p>
<pre><code>bar
</code></pre>
<p>baz</p>
<blockquote>
<p>bam</p>
</blockquote>
</li>
</ol>
```

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

Example 264

```
-·Foo

······bar


······baz
```

```html
<ul>
<li>
<p>Foo</p>
<pre><code>bar


baz
</code></pre>
</li>
</ul>
```

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

Example 265

```
123456789.·ok
```

```html
<ol·start="123456789">
<li>ok</li>
</ol>
```

Example 266

```
1234567890.·not·ok
```

```html
<p>1234567890.·not·ok</p>
```

A start number may begin with 0s:

Example 267

```
0.·ok
```

```
<ol·start="0">
<li>ok</li>
</ol>
```

Example 268

```
003.·ok
```

```
<ol·start="3">
<li>ok</li>
</ol>
```

A start number may not be negative:

Example 269

```
-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 of indentation, 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 preceded by four spaces of indentation beyond the edge of the region where text will be included in the list item. In the following case that is 6 spaces:

Example 270

```
-·foo

······bar
```

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

And in this case it is 11 spaces:

Example 271

```
··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 preceded by *one* space of indentation after the list marker:

Example 272

```
····indented·code

paragraph

····more·code
```

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

Example 273

```
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 of indentation is interpreted as space inside the code block:

Example 274

```
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 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 character other than a space or tab, and (b) cases in which they begin with an indented code block. In a case like the following, where the first block begins with three spaces of indentation, the rules do not allow us to form a list item by indenting the whole thing and prepending a list marker:

Example 275

```
···foo

bar
```

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

Example 276

```
-····foo

··bar
```

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

This is not a significant restriction, because when a block is preceded by up to three spaces of indentation, the indentation can always be removed without a change in interpretation, allowing rule #1 to be applied. So, in the above case:

Example 277

```
-··foo

···bar
```

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

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*, and *M* is a list marker of width *W*, then the result of prepending *M* to the first line of *Ls*, and preceding subsequent lines of *Ls* by *W + 1* spaces of indentation, 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 278

````
-
··foo
-
··```
··bar
··```
-
······baz
````

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

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

Example 279

```
-···
··foo
```

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

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

Example 280

```
-

··foo
```

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

Here is an empty bullet list item:

Example 281

```
-·foo
-
-·bar
```

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

It does not matter whether there are spaces or tabs following the list marker:

Example 282

```
-·foo
-···
-·bar
```

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

Here is an empty ordered list item:

Example 283

```
1.·foo
2.
3.·bar
```

```
<ol>
<li>foo</li>
<li></li>
<li>bar</li>
</ol>
```

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

Example 284

```
*
```

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

However, an empty list item cannot interrupt a paragraph:

Example 285

```
foo
*

foo
1.
```

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

4. **Indentation.** If a sequence of lines *Ls* constitutes a list item according to rule #1, #2, or #3, then the result of preceding each line of *Ls* by up to three spaces of indentation (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 286

```
·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 two spaces:

Example 287

```
··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 288

```
···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 289

```
····1.··A·paragraph
········with·two·lines.

············indented·code

········>·A·block·quote.
```

```
<pre><code>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 character other than a space or tab after the indentation is paragraph continuation text is a list item with the same contents and attributes. The unindented lines are called **lazy continuation line**s.

Here is an example with lazy continuation lines:

Example 290

```
··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 291

```
··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 292

```
>·1.·>·Blockquote
continued·here.
```

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

Example 293

```
>·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 of indentation 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 294

```
-·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 295

```
-·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 296

```
10)·foo
····-·bar
```

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

Three is not enough:

Example 297

```
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 298

```
-·-·foo
```

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

Example 299

```
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 300

```
-·#·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 up to three spaces of indentation for 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 301

```
-·foo
-·bar
+·baz
```

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

Example 302

```
1.·foo
2.·bar
3)·baz
```

```
<ol>
<li>foo</li>
<li>bar</li>
</ol>
<ol·start="3">
<li>baz</li>
</ol>
```

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

Example 303

```
Foo
-·bar
-·baz
```

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

`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 the problem of unwanted lists in paragraphs with hard-wrapped numerals, we allow only lists starting with 1 to interrupt paragraphs. Thus,

Example 304

```
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 305

```
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 306

```
-·foo

-·bar


-·baz
```

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

Example 307

```
-·foo
··-·bar
····-·baz


······bim
```

```
<ul>
<li>foo
<ul>
<li>bar
<ul>
<li>
<p>baz</p>
<p>bim</p>
</li>
</ul>
</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 308

```
-·foo
-·bar

<!--·-->

-·baz
-·bim
```

```
<ul>
<li>foo</li>
<li>bar</li>
</ul>
<!--·-->
<ul>
<li>baz</li>
<li>bim</li>
</ul>
```

Example 309

```
-···foo

····notcode

-···foo

<!--·-->

····code
```

```
<ul>
<li>
<p>foo</p>
<p>notcode</p>
</li>
<li>
<p>foo</p>
</li>
</ul>
<!--·-->
<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 310

```
-·a
·-·b
··-·c
···-·d
··-·e
·-·f
-·g
```

```
<ul>
<li>a</li>
<li>b</li>
<li>c</li>
<li>d</li>
<li>e</li>
<li>f</li>
<li>g</li>
</ul>
```

Example 311

```
1.·a

··2.·b

···3.·c
```

```
<ol>
<li>
<p>a</p>
</li>
<li>
<p>b</p>
</li>
<li>
<p>c</p>
</li>
</ol>
```

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

Example 312

```
-·a                                     <ul>
·-·b                                    <li>a</li>
··-·c                                   <li>b</li>
···-·d                                  <li>c</li>
····-·e                                 <li>d
                                        -·e</li>
                                        </ul>
```

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

Example 313

```
1.·a                                    <ol>
                                        <li>
··2.·b                                  <p>a</p>
                                        </li>
····3.·c                                <li>
                                        <p>b</p>
                                        </li>
                                        </ol>
                                        <pre><code>3.·c
                                        </code></pre>
```

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

Example 314

```
-·a                                     <ul>
-·b                                     <li>
                                        <p>a</p>
-·c                                     </li>
                                        <li>
                                        <p>b</p>
                                        </li>
                                        <li>
                                        <p>c</p>
                                        </li>
                                        </ul>
```

So is this, with a empty second item:

Example 315

```
*·a
*

*·c
```

```
<ul>
<li>
<p>a</p>
</li>
<li></li>
<li>
<p>c</p>
</li>
</ul>
```

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

Example 316

```
-·a
-·b

··c
-·d
```

```
<ul>
<li>
<p>a</p>
</li>
<li>
<p>b</p>
<p>c</p>
</li>
<li>
<p>d</p>
</li>
</ul>
```

Example 317

```
-·a
-·b

··[ref]:·/url
-·d
```

```
<ul>
<li>
<p>a</p>
</li>
<li>
<p>b</p>
</li>
<li>
<p>d</p>
</li>
</ul>
```

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

Example 318

```
-·a
-·```
··b


··```
-·c
```

```
<ul>
<li>a</li>
<li>
<pre><code>b


</code></pre>
</li>
<li>c</li>
</ul>
```

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 319

```
-·a
··-·b

····c
-·d
```

```
<ul>
<li>a
<ul>
<li>
<p>b</p>
<p>c</p>
</li>
</ul>
</li>
<li>d</li>
</ul>
```

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

Example 320

```
*·a
··>·b
··>
*·c
```

```
<ul>
<li>a
<blockquote>
<p>b</p>
</blockquote>
</li>
<li>c</li>
</ul>
```

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

Example 321

```
- a
  > b
  ```
  c
  ```
- d
```

```
<ul>
<li>a
<blockquote>
<p>b</p>
</blockquote>
<pre><code>c
</code></pre>
</li>
<li>d</li>
</ul>
```

A single-paragraph list is tight:

Example 322

```
- a
```

```
<ul>
<li>a</li>
</ul>
```

Example 323

```
- a
  - b
```

```
<ul>
<li>a
<ul>
<li>b</li>
</ul>
</li>
</ul>
```

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

Example 324

~~~
1. ```
   foo
   ```

   bar
~~~

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

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

Example 325

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

Example 326

```
-·a                                     <ul>
··-·b                                   <li>
··-·c                                   <p>a</p>
                                        <ul>
-·d                                     <li>b</li>
··-·e                                   <li>c</li>
··-·f                                   </ul>
                                        </li>
                                        <li>
                                        <p>d</p>
                                        <ul>
                                        <li>e</li>
                                        <li>f</li>
                                        </ul>
                                        </li>
                                        </ul>
```

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 327

```
`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 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 these 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;
</code>&quot;&gt;`</p>
```

But this is an HTML tag:

Example 344

```
<a·href="`">`
```

```
<p><a·href="`">`</p>
```

And this is code:

Example 345

```
`<https://foo.bar.`baz>`
```

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

But this is an autolink:

Example 346

```
<https://foo.bar.`baz>`
```

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

6.2 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 Unicode punctuation character, or (2b) followed by a Unicode punctuation character and preceded by Unicode whitespace or a Unicode 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 Unicode punctuation character, or (2b) preceded by a Unicode punctuation character and followed by Unicode whitespace or a Unicode 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 a Unicode punctuation character.

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 a Unicode punctuation character.

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 a Unicode punctuation character.
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 a Unicode punctuation character.
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><em>foo·bar</em></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*&quot;foo&quot;*</p>
```

Unicode nonbreaking spaces count as whitespace, too:

Example 353

```
* a *
```

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

Unicode symbols count as punctuation, too:

Example 354

```
*$*alpha.

*£*bravo.

*€*charlie.
```

```
<p>*$*alpha.</p>
<p>*£*bravo.</p>
<p>*€*charlie.</p>
```

Intraword emphasis with `*` is permitted:

Example 355

```
foo*bar*
```

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

Example 356

```
5*6*78
```

```
<p>5<em>6</em>78</p>
```

Rule 2:

Example 357

```
_foo bar_
```

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

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

Example 358

```
_ foo bar_
```

```
<p>_ foo bar_</p>
```

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

Example 359

```
a_"foo"_
```

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

Emphasis with _ is not allowed inside words:

Example 360

```
foo_bar_
```

```
<p>foo_bar_</p>
```

Example 361

```
5_6_78
```

```
<p>5_6_78</p>
```

Example 362

```
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 363

```
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 364

```
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 365

```
_foo*
```

```
<p>_foo*</p>
```

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

Example 366

```
*foo·bar·*
```

```
<p>*foo·bar·*</p>
```

A line ending also counts as whitespace:

Example 367

```
*foo·bar
*
```

```
<p>*foo·bar
*</p>
```

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 368

```
*(*foo)
```

```
<p>*(*foo)</p>
```

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

Example 369

```
*(*foo*)*
```

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

Intraword emphasis with `*` is allowed:

Example 370

```
*foo*bar
```

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

Rule 4:

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

Example 371

```
_foo·bar·_                              <p>_foo·bar·_</p>
```

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

Example 372

```
_(_foo)                                 <p>_(_foo)</p>
```

This is emphasis within emphasis:

Example 373

```
_(_foo_)_                               <p><em>(<em>foo</em>)</em></p>
```

Intraword emphasis is disallowed for _:

Example 374

```
_foo_bar                                <p>_foo_bar</p>
```

Example 375

```
_пристаням_стремятся                    <p>_пристаням_стремятся</p>
```

Example 376

```
_foo_bar_baz_                           <p><em>foo_bar_baz</em></p>
```

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

Example 377

```
_(bar)_.                                <p><em>(bar)</em>.</p>
```

Rule 5:

Example 378

```
**foo·bar**                             <p><strong>foo·bar</strong></p>
```

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

Example 379

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

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 380

```
a**"foo"**
```

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

Intraword strong emphasis with `**` is permitted:

Example 381

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

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

Rule 6:

Example 382

```
__foo·bar__
```

```
<p><strong>foo·bar</strong></p>
```

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

Example 383

```
__·foo·bar__
```

```
<p>__·foo·bar__</p>
```

A line ending counts as whitespace:

Example 384

```
__
foo·bar__
```

```
<p>__
foo·bar__</p>
```

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

Example 385

```
a__"foo"__
```

```
<p>a__&quot;foo&quot;__</p>
```

Intraword strong emphasis is forbidden with `__`:

Example 386

```
foo__bar__
```

```
<p>foo__bar__</p>
```

Example 387

```
5__6__78
```

```
<p>5__6__78</p>
```

Example 388

```
пристаням__стремятся__                  <p>пристаням__стремятся__</p>
```

Example 389

```
__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 390

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

Rule 7:

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

Example 391

```
**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 392

```
**(**foo)                               <p>**(**foo)</p>
```

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

Example 393

```
*(**foo**)*                             <p><em>(<strong>foo</strong>)</em>
                                        </p>
```

Example 394

```
**Gomphocarpus·(*Gomphocarpus·          <p><strong>Gomphocarpus·
physocarpus*,·syn.                      (<em>Gomphocarpus·physocarpus</em>,·
*Asclepias·physocarpa*)**               syn.
                                        <em>Asclepias·physocarpa</em>)
                                        </strong></p>
```

Example 395

`**foo·"*bar*"·foo**`	`<p><strong>foo·&quot;<em>bar</em>&quot;·foo</strong></p>`

Intraword emphasis:

Example 396

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

Rule 8:

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

Example 397

`__foo·bar·__`	`<p>__foo·bar·__</p>`

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

Example 398

`__(__foo)`	`<p>__(__foo)</p>`

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

Example 399

`_(__foo__)_`	`<p><em>(<strong>foo</strong>)</em></p>`

Intraword strong emphasis is forbidden with `__`:

Example 400

`__foo__bar`	`<p>__foo__bar</p>`

Example 401

`__пристаням__стремятся`	`<p>__пристаням__стремятся</p>`

Example 402

`__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 403

```
__(bar)__.
```

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

Rule 9:

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

Example 404

```
*foo·[bar](/url)*
```

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

Example 405

```
*foo
bar*
```

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

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

Example 406

```
_foo·__bar__·baz_
```

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

Example 407

```
_foo·_bar_·baz_
```

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

Example 408

```
__foo_·bar_
```

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

Example 409

```
*foo·*bar**
```

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

Example 410

```
*foo·**bar**·baz*
```

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

Example 411

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

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

Note that in the preceding case, the interpretation

```
<p><em>foo</em><em>bar<em></em>baz</em></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 412

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

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

The same condition ensures that the following cases are all strong emphasis nested inside emphasis, even when the interior whitespace is omitted:

Example 413

```
***foo**·bar*
```

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

Example 414

```
*foo·**bar***
```

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

Example 415

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

```
<p><em>foo<strong>bar</strong></em>
</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 416

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

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

Example 417

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

```
<p>foo<strong><strong>
<strong>bar</strong></strong>
</strong>***baz</p>
```

Indefinite levels of nesting are possible:

Example 418

```
*foo·**bar·*baz*·bim**·bop*
```

```
<p><em>foo·<strong>bar·<em>baz</em>·
bim</strong>·bop</em></p>
```

Example 419

```
*foo·[*bar*](/url)*
```

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

There can be no empty emphasis or strong emphasis:

Example 420

```
**·is·not·an·empty·emphasis
```

```
<p>**·is·not·an·empty·emphasis</p>
```

Example 421

```
****·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 422

```
**foo·[bar](/url)**
```

```
<p><strong>foo·<a·href="/url">bar</a>
</strong></p>
```

Example 423

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

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

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

Example 424

```
__foo·_bar_·baz__
```

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

Example 425

```
__foo·__bar__·baz__
```

```
<p><strong>foo·<strong>bar</strong>·
baz</strong></p>
```

Example 426

```
____foo__·bar__
```

```
<p><strong><strong>foo</strong>·
bar</strong></p>
```

Example 427

```
**foo·**bar****
```

```
<p><strong>foo·<strong>bar</strong>
</strong></p>
```

Example 428

```
**foo·*bar*·baz**
```

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

Example 429

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

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

Example 430

```
***foo*·bar**
```

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

Example 431

```
**foo·*bar***
```

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

Indefinite levels of nesting are possible:

Example 432

```
**foo·*bar·**baz**
bim*·bop**
```

```
<p><strong>foo·<em>bar·
<strong>baz</strong>
bim</em>·bop</strong></p>
```

Example 433

```
**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 434

```
__·is·not·an·empty·emphasis
```

```
<p>__·is·not·an·empty·emphasis</p>
```

Example 435

```
____·is·not·an·empty·strong·emphasis
```

```
<p>____·is·not·an·empty·strong·
emphasis</p>
```

Rule 11:

Example 436

```
foo·***
```

```
<p>foo·***</p>
```

Example 437

```
foo·*\**                                <p>foo·<em>*</em></p>
```

Example 438

```
foo·*_*                                 <p>foo·<em>_</em></p>
```

Example 439

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

Example 440

```
foo·**\***                              <p>foo·<strong>*</strong></p>
```

Example 441

```
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 442

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

Example 443

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

Example 444

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

Example 445

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

Example 446

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

Example 447

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

Rule 12:

Example 448

```
foo·___                                 <p>foo·___</p>
```

Example 449

```
foo·_\__
```

```
<p>foo·<em>_</em></p>
```

Example 450

```
foo·_*_
```

```
<p>foo·<em>*</em></p>
```

Example 451

```
foo·_____
```

```
<p>foo·_____</p>
```

Example 452

```
foo·__\___
```

```
<p>foo·<strong>_</strong></p>
```

Example 453

```
foo·__*__
```

```
<p>foo·<strong>*</strong></p>
```

Example 454

```
__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 455

```
_foo__
```

```
<p><em>foo</em>_</p>
```

Example 456

```
___foo__
```

```
<p>_<strong>foo</strong></p>
```

Example 457

```
____foo_
```

```
<p>___<em>foo</em></p>
```

Example 458

```
__foo___
```

```
<p><strong>foo</strong>_</p>
```

Example 459

```
_foo____
```

```
<p><em>foo</em>___</p>
```

Rule 13 implies that if you want emphasis nested directly inside emphasis, you must use different delimiters:

Example 460

```
**foo**
```

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

Example 461

```
*_foo_*
```

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

Example 462

```
__foo__
```

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

Example 463

```
_*foo*_
```

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

However, strong emphasis within strong emphasis is possible without switching delimiters:

Example 464

```
****foo****
```

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

Example 465

```
____foo____
```

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

Rule 13 can be applied to arbitrarily long sequences of delimiters:

Example 466

```
******foo******
```

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

Rule 14:

Example 467

```
***foo***
```

```
<p><em><strong>foo</strong></em></p>
```

Example 468

```
_____foo_____
```

```
<p><em><strong><strong>foo</strong>
</strong></em></p>
```

Rule 15:

Example 469

```
*foo·_bar*·baz_
```

```
<p><em>foo·_bar</em>·baz_</p>
```

Example 470

```
*foo·__bar·*baz·bim__·bam*
```

```
<p><em>foo·<strong>bar·*baz·
bim</strong>·bam</em></p>
```

Rule 16:

Example 471

```
**foo·**bar·baz**
```

```
<p>**foo·<strong>bar·baz</strong></p>
```

Example 472

```
*foo·*bar·baz*
```

```
<p>*foo·<em>bar·baz</em></p>
```

Rule 17:

Example 473

```
*[bar*](/url)
```

```
<p>*<a·href="/url">bar*</a></p>
```

Example 474

```
_foo·[bar_](/url)
```

```
<p>_foo·<a·href="/url">bar_</a></p>
```

Example 475

```
*<img·src="foo"·title="*"/>
```

```
<p>*<img·src="foo"·title="*"/></p>
```

Example 476

```
**<a·href="**">
```

```
<p>**<a·href="**"></p>
```

Example 477

```
__<a·href="__">
```

```
<p>__<a·href="__"></p>
```

Example 478

```
*a·`*`*
```

```
<p><em>a·<code>*</code></em></p>
```

Example 479

```
_a·`_`_
```

```
<p><em>a·<code>_</code></em></p>
```

Example 480

```
**a<https://foo.bar/?q=**>
```

```
<p>**a<a·href="https://foo.bar/?
q=**">https://foo.bar/?q=**</a></p>
```

Example 481

```
__a<https://foo.bar/?q=__>
```

```
<p>__a<a·href="https://foo.bar/?
q=__">https://foo.bar/?q=__</a></p>
```

6.3 Links

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 endings or unescaped `<` or `>` characters, or

- a nonempty sequence of characters that does not start with `<`, does not include ASCII control characters or space character, 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 `(`, an optional link destination, an optional link title, and a right parenthesis `)`. These four components may be separated by spaces, tabs, and up to one line ending. If both link destination and link title are present, they *must* be separated by spaces, tabs, and up to one line ending.

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 482

```
[link](/uri·"title")
```

```
<p><a·href="/uri"·
title="title">link</a></p>
```

The title, the link text and even the destination may be omitted:

Example 483

```
[link](/uri)
```

```
<p><a·href="/uri">link</a></p>
```

Example 484

```
[](./target.md)
```

```
<p><a·href="./target.md"></a></p>
```

Example 485

```
[link]()
```

```
<p><a·href="">link</a></p>
```

Example 486

```
[link](<>)
```

```
<p><a·href="">link</a></p>
```

Example 487

```
[]()
```

```
<p><a·href=""></a></p>
```

The destination can only contain spaces if it is enclosed in pointy brackets:

Example 488

```
[link](/my·uri)
```

```
<p>[link](/my·uri)</p>
```

Example 489

```
[link](</my·uri>)
```

```
<p><a·href="/my%20uri">link</a></p>
```

The destination cannot contain line endings, even if enclosed in pointy brackets:

Example 490

```
[link](foo
bar)
```

```
<p>[link](foo
bar)</p>
```

Example 491

```
[link](<foo
bar>)
```

```
<p>[link](<foo
bar>)</p>
```

The destination can contain `)` if it is enclosed in pointy brackets:

Example 492

```
[a](<b)c>)
```

```
<p><a href="b)c">a</a></p>
```

Pointy brackets that enclose links must be unescaped:

Example 493

```
[link](<foo\>)
```

```
<p>[link](&lt;foo&gt;)</p>
```

These are not links, because the opening pointy bracket is not matched properly:

Example 494

```
[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 495

```
[link](\(foo\))
```

```
<p><a href="(foo)">link</a></p>
```

Any number of parentheses are allowed without escaping, as long as they are balanced:

Example 496

```
[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 497

```
[link](foo(and(bar))
```

```
<p>[link](foo(and(bar))</p>
```

Example 498

```
[link](foo\(and\(bar\))
```

```
<p><a·href="foo(and(bar)">link</a></p>
```

Example 499

```
[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 500

```
[link](foo\)\:)
```

```
<p><a·href="foo):">link</a></p>
```

A link can contain fragment identifiers and queries:

Example 501

```
[link](#fragment)

[link](https://example.com#fragment)

[link](https://example.com?foo=3#frag)
```

```
<p><a·href="#fragment">link</a></p>
<p><a·href="https://example.com#fragment">link</a></p>
<p><a·href="https://example.com?foo=3#frag">link</a></p>
```

Note that a backslash before a non-escapable character is just a backslash:

Example 502

```
[link](foo\bar)
```

```
<p><a·href="foo%5Cbar">link</a></p>
```

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 503

```
[link](foo%20b&auml;)
```

```
<p><a·href="foo%20b%C3%A4">link</a></p>
```

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 504

```
[link]("title")
```

```
<p><a·href="%22title%22">link</a></p>
```

Titles may be in single quotes, double quotes, or parentheses:

Example 505

```
[link](/url·"title")
[link](/url·'title')
[link](/url·(title))
```

```
<p><a·href="/url"·
title="title">link</a>
<a·href="/url"·title="title">link</a>
<a·href="/url"·title="title">link</a>
</p>
```

Backslash escapes and entity and numeric character references may be used in titles:

Example 506

```
[link](/url·"title·\"&quot;")
```

```
<p><a·href="/url"·title="title·
&quot;&quot;">link</a></p>
```

Titles must be separated from the link using spaces, tabs, and up to one line ending. Other Unicode whitespace like non-breaking space doesn't work.

Example 507

```
[link](/url "title")
```

```
<p><a·
href="/url%C2%A0%22title%22">link</a>
</p>
```

Nested balanced quotes are not allowed without escaping:

Example 508

```
[link](/url·"title·"and"·title")
```

```
<p>[link](/url·&quot;title·
&quot;and&quot;·title&quot;)</p>
```

But it is easy to work around this by using a different quote type:

Example 509

```
[link](/url·'title·"and"·title')
```

```
<p><a·href="/url"·title="title·
&quot;and&quot;·title">link</a></p>
```

(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.)

Spaces, tabs, and up to one line ending is allowed around the destination and title:

Example 510

```
[link](···/uri
··"title"··)
```

```
<p><a·href="/uri"·
title="title">link</a></p>
```

But it is not allowed between the link text and the following parenthesis:

Example 511

```
[link]·(/uri)
```

```
<p>[link]·(/uri)</p>
```

The link text may contain balanced brackets, but not unbalanced ones, unless they are escaped:

Example 512

```
[link·[foo·[bar]]](/uri)
```

```
<p><a·href="/uri">link·[foo·[bar]]
</a></p>
```

Example 513

```
[link]·bar](/uri)
```

```
<p>[link]·bar](/uri)</p>
```

Example 514

```
[link·[bar](/uri)
```

```
<p>[link·<a·href="/uri">bar</a></p>
```

Example 515

```
[link·\[bar](/uri)
```

```
<p><a·href="/uri">link·[bar</a></p>
```

The link text may contain inline content:

Example 516

```
[link·*foo·**bar**·`#`*](/uri)
```

```
<p><a·href="/uri">link·<em>foo·
<strong>bar</strong>·<code>#</code>
</em></a></p>
```

Example 517

```
[![moon](moon.jpg)](/uri)
```

```
<p><a·href="/uri"><img·src="moon.jpg"·
alt="moon"·/></a></p>
```

However, links may not contain other links, at any level of nesting.

Example 518

```
[foo·[bar](/uri)](/uri)
```

```
<p>[foo·<a·href="/uri">bar</a>](/uri)
</p>
```

Example 519

```
[foo·*[bar·[baz](/uri)](/uri)*](/uri)
```

```
<p>[foo·<em>[bar·<a·
href="/uri">baz</a>](/uri)</em>]
(/uri)</p>
```

Example 520

```
![[[foo](uri1)](uri2)](uri3)
```

```
<p><img·src="uri3"·alt="[foo](uri2)"·
/></p>
```

These cases illustrate the precedence of link text grouping over emphasis grouping:

Example 521

```
*[foo*](/uri)
```

```
<p>*<a·href="/uri">foo*</a></p>
```

Example 522

```
[foo·*bar](baz*)
```

```
<p><a·href="baz*">foo·*bar</a></p>
```

Note that brackets that *aren’t* part of links do not take precedence:

Example 523

```
*foo·[bar*·baz]
```

```
<p><em>foo·[bar</em>·baz]</p>
```

These cases illustrate the precedence of HTML tags, code spans, and autolinks over link grouping:

Example 524

```
[foo·<bar·attr="](baz)">
```

```
<p>[foo·<bar·attr="](baz)"></p>
```

Example 525

```
[foo`](/uri)`
```

```
<p>[foo<code>](/uri)</code></p>
```

Example 526

```
[foo<https://example.com/?search=]
(uri)>
```

```
<p>[foo<a·href="https://example.com/?
search=%5D(uri)">https://example.com/
?search=](uri)</a></p>
```

There are three kinds of **reference link**s: 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 character that is not a space, tab, or line ending. 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 spaces, tabs, and line endings, and collapse consecutive internal spaces, tabs, and line endings 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 link's URI and title are provided by the matching link reference definition.

Here is a simple example:

Example 527

```
[foo][bar]

[bar]:·/url·"title"
```

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

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 528

```
[link·[foo·[bar]]][ref]

[ref]:·/uri
```

```
<p><a·href="/uri">link·[foo·[bar]]
</a></p>
```

Example 529

```
[link·\[bar][ref]

[ref]:·/uri
```

```
<p><a·href="/uri">link·[bar</a></p>
```

The link text may contain inline content:

Example 530

```
[link·*foo·**bar**·`#`*][ref]

[ref]:·/uri
```

```
<p><a·href="/uri">link·<em>foo·
<strong>bar</strong>·<code>#</code>
</em></a></p>
```

Example 531

```
[![moon](moon.jpg)][ref]

[ref]:·/uri
```

```
<p><a·href="/uri"><img·src="moon.jpg"·
alt="moon"·/></a></p>
```

However, links may not contain other links, at any level of nesting.

Example 532

```
[foo·[bar](/uri)][ref]

[ref]:·/uri
```

```
<p>[foo·<a·href="/uri">bar</a>]<a·
href="/uri">ref</a></p>
```

Example 533

```
[foo·*bar·[baz][ref]*][ref]

[ref]:·/uri
```

```
<p>[foo·<em>bar·<a·
href="/uri">baz</a></em>]<a·
href="/uri">ref</a></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 534

```
*[foo*][ref]

[ref]:·/uri
```

```
<p>*<a·href="/uri">foo*</a></p>
```

Example 535

```
[foo·*bar][ref]*

[ref]:·/uri
```

```
<p><a·href="/uri">foo·*bar</a>*</p>
```

These cases illustrate the precedence of HTML tags, code spans, and autolinks over link grouping:

Example 536

```
[foo·<bar·attr="][ref]">

[ref]:·/uri
```

```
<p>[foo·<bar·attr="][ref]"></p>
```

Example 537

```
[foo`][ref]`

[ref]:·/uri
```

```
<p>[foo<code>][ref]</code></p>
```

Example 538

```
[foo<https://example.com/?search=]
[ref]>

[ref]:·/uri
```

```
<p>[foo<a·href="https://example.com/?
search=%5D%5Bref%5D">https://example.
com/?search=][ref]</a></p>
```

Matching is case-insensitive:

[Example 539]

```
[foo][BaR]

[bar]:·/url·"title"
```

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

Unicode case fold is used:

[Example 540]

```
[ß]

[SS]:·/url
```

```
<p><a·href="/url">ß</a></p>
```

Consecutive internal spaces, tabs, and line endings are treated as one space for purposes of determining matching:

[Example 541]

```
[Foo
··bar]:·/url

[Baz][Foo·bar]
```

```
<p><a·href="/url">Baz</a></p>
```

No spaces, tabs, or line endings are allowed between the link text and the link label:

[Example 542]

```
[foo]·[bar]

[bar]:·/url·"title"
```

```
<p>[foo]·<a·href="/url"·
title="title">bar</a></p>
```

[Example 543]

```
[foo]
[bar]

[bar]:·/url·"title"
```

```
<p>[foo]
<a·href="/url"·title="title">bar</a>
</p>
```

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 544

```
[foo]:·/url1

[foo]:·/url2

[bar][foo]
```

```
<p><a·href="/url1">bar</a></p>
```

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 545

```
[bar][foo\!]

[foo!]:·/url
```

```
<p>[bar][foo!]</p>
```

Link labels cannot contain brackets, unless they are backslash-escaped:

Example 546

```
[foo][ref[]

[ref[]:·/uri
```

```
<p>[foo][ref[]</p>
<p>[ref[]:·/uri</p>
```

Example 547

```
[foo][ref[bar]]

[ref[bar]]:·/uri
```

```
<p>[foo][ref[bar]]</p>
<p>[ref[bar]]:·/uri</p>
```

Example 548

```
[[[foo]]]

[[[foo]]]:·/url
```

```
<p>[[[foo]]]</p>
<p>[[[foo]]]:·/url</p>
```

Example 549

```
[foo][ref\[]

[ref\[]:·/uri
```

```
<p><a·href="/uri">foo</a></p>
```

Note that in this example `]` is not backslash-escaped:

Example 550

```
[bar\\]:·/uri

[bar\\]
```

```
<p><a·href="/uri">bar\</a></p>
```

A link label must contain at least one character that is not a space, tab, or line ending:

Example 551

```
[]

[]:·/uri
```

```
<p>[]</p>
<p>[]:·/uri</p>
```

Example 552

```
[
·]

[
·]:·/uri
```

```
<p>[
]</p>
<p>[
]:·/uri</p>
```

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 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 553

```
[foo][]

[foo]:·/url·"title"
```

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

Example 554

```
[*foo*·bar][]

[*foo*·bar]:·/url·"title"
```

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

The link labels are case-insensitive:

Example 555

```
[Foo][]

[foo]:·/url·"title"
```

```
<p><a·href="/url"·
title="title">Foo</a></p>
```

As with full reference links, spaces, tabs, or line endings are not allowed between the two sets of brackets:

Example 556

```
[foo]·
[]

[foo]:·/url·"title"
```

```
<p><a·href="/url"·
title="title">foo</a>
[]</p>
```

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 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 557

```
[foo]

[foo]:·/url·"title"
```

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

Example 558

```
[*foo*·bar]

[*foo*·bar]:·/url·"title"
```

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

Example 559

```
[[*foo*·bar]]

[*foo*·bar]:·/url·"title"
```

```
<p>[<a·href="/url"·title="title">
<em>foo</em>·bar</a>]</p>
```

Example 560

```
[[bar·[foo]

[foo]:·/url
```

```
<p>[[bar·<a·href="/url">foo</a></p>
```

The link labels are case-insensitive:

Example 561

```
[Foo]

[foo]:·/url·"title"
```

```
<p><a·href="/url"·
title="title">Foo</a></p>
```

A space after the link text should be preserved:

Example 562

```
[foo]·bar

[foo]:·/url
```

```
<p><a·href="/url">foo</a>·bar</p>
```

If you just want bracketed text, you can backslash-escape the opening bracket to avoid links:

Example 563

```
\[foo]

[foo]:·/url·"title"
```

```
<p>[foo]</p>
```

Note that this is a link, because a link label ends with the first following closing bracket:

Example 564

```
[foo*]:·/url

*[foo*]
```

```
<p>*<a·href="/url">foo*</a></p>
```

Full and collapsed references take precedence over shortcut references:

Example 565

```
[foo][bar]

[foo]:·/url1
[bar]:·/url2
```

```
<p><a·href="/url2">foo</a></p>
```

Example 566

```
[foo][]

[foo]:·/url1
```

```
<p><a·href="/url1">foo</a></p>
```

Inline links also take precedence:

Example 567

```
[foo]()

[foo]:·/url1
```

```
<p><a·href="">foo</a></p>
```

Example 568

```
[foo](not·a·link)

[foo]:·/url1
```

```
<p><a·href="/url1">foo</a>(not·a·
link)</p>
```

In the following case `[bar][baz]` is parsed as a reference, `[foo]` as normal text:

Example 569

```
[foo][bar][baz]

[baz]:·/url
```

```
<p>[foo]<a·href="/url">bar</a></p>
```

Here, though, `[foo][bar]` is parsed as a reference, since `[bar]` is defined:

Example 570

```
[foo][bar][baz]

[baz]:·/url1
[bar]:·/url2
```

```
<p><a·href="/url2">foo</a><a·
href="/url1">baz</a></p>
```

Here `[foo]` is not parsed as a shortcut reference, because it is followed by a link label (even though `[bar]` is not defined):

Example 571

```
[foo][bar][baz]

[baz]:·/url1
[foo]:·/url2
```

```
<p>[foo]<a·href="/url1">bar</a></p>
```

6.4 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 572

```
![foo](/url·"title")
```

```
<p><img·src="/url"·alt="foo"·
title="title"·/></p>
```

Example 573

```
![foo·*bar*]

[foo·*bar*]:·train.jpg·"train·&·
tracks"
```

```
<p><img·src="train.jpg"·alt="foo·bar"·
title="train·&amp;·tracks"·/></p>
```

Example 574

```
![foo·![bar](/url)](/url2)
```

```
<p><img·src="/url2"·alt="foo·bar"·/>
</p>
```

Example 575

```
![foo·[bar](/url)](/url2)
```

```
<p><img·src="/url2"·alt="foo·bar"·/>
</p>
```

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 576

```
![foo·*bar*][]

[foo·*bar*]:·train.jpg·"train·&·
tracks"
```

```
<p><img·src="train.jpg"·alt="foo·bar"·
title="train·&amp;·tracks"·/></p>
```

Example 577

```
![foo·*bar*][foobar]

[FOOBAR]:·train.jpg·"train·&·tracks"
```

```
<p><img·src="train.jpg"·alt="foo·bar"·
title="train·&amp;·tracks"·/></p>
```

Example 578

```
![foo](train.jpg)
```

```
<p><img·src="train.jpg"·alt="foo"·/>
</p>
```

Example 579

```
My·![foo·bar](/path/to/train.jpg··
"title"···)
```

```
<p>My·<img·src="/path/to/train.jpg"·
alt="foo·bar"·title="title"·/></p>
```

Example 580

```
![foo](<url>)
```

```
<p><img·src="url"·alt="foo"·/></p>
```

Example 581

```
![](/url)
```

```
<p><img·src="/url"·alt=""·/></p>
```

Reference-style:

Example 582

```
![foo][bar]

[bar]:·/url
```

```
<p><img·src="/url"·alt="foo"·/></p>
```

Example 583

```
![foo][bar]

[BAR]:·/url
```

```
<p><img·src="/url"·alt="foo"·/></p>
```

Collapsed:

Example 584

```
![foo][]

[foo]:·/url·"title"
```

```
<p><img·src="/url"·alt="foo"·
title="title"·/></p>
```

Example 585

```
![*foo*·bar][]

[*foo*·bar]:·/url·"title"
```

```
<p><img·src="/url"·alt="foo·bar"·
title="title"·/></p>
```

The labels are case-insensitive:

Example 586

```
![Foo][]

[foo]:·/url·"title"
```

```
<p><img·src="/url"·alt="Foo"·
title="title"·/></p>
```

As with reference links, spaces, tabs, and line endings, are not allowed between the two sets of brackets:

Example 587

```
![foo]·
[]

[foo]:·/url·"title"
```

```
<p><img·src="/url"·alt="foo"·
title="title"·/>
[]</p>
```

Shortcut:

Example 588

```
![foo]

[foo]:·/url·"title"
```

```
<p><img·src="/url"·alt="foo"·
title="title"·/></p>
```

Example 589

```
![*foo*·bar]

[*foo*·bar]:·/url·"title"
```

```
<p><img·src="/url"·alt="foo·bar"·
title="title"·/></p>
```

Note that link labels cannot contain unescaped brackets:

Example 590

```
![[foo]]

[[foo]]:·/url·"title"
```

```
<p>![[foo]]</p>
<p>[[foo]]:·/url·&quot;title&quot;
</p>
```

The link labels are case-insensitive:

Example 591

```
![Foo]

[foo]:·/url·"title"
```

```
<p><img·src="/url"·alt="Foo"·
title="title"·/></p>
```

If you just want a literal `!` followed by bracketed text, you can backslash-escape the opening `[`:

Example 592

```
!\[foo]

[foo]:·/url·"title"
```

```
<p>![foo]</p>
```

If you want a link after a literal `!`, backslash-escape the `!`:

Example 593

```
\![foo]

[foo]:·/url·"title"
```

```
<p>!<a·href="/url"·
title="title">foo</a></p>
```

6.5 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 control characters, space, `<`, 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 594

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

```
<p><a·
href="http://foo.bar.baz">http://foo.
bar.baz</a></p>
```

Example 595

```
<https://foo.bar.baz/test?
q=hello&id=22&boolean>
```

```
<p><a·href="https://foo.bar.baz/test?
q=hello&amp;id=22&amp;boolean">https:
//foo.bar.baz/test?
q=hello&amp;id=22&amp;boolean</a></p>
```

Example 596

```
<irc://foo.bar:2233/baz>
```

```
<p><a·
href="irc://foo.bar:2233/baz">irc://f
oo.bar:2233/baz</a></p>
```

Uppercase is also fine:

Example 597

```
<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 598

```
<a+b+c:d>
```

```
<p><a·href="a+b+c:d">a+b+c:d</a></p>
```

Example 599

```
<made-up-scheme://foo,bar>
```

```
<p><a·href="made-up-
scheme://foo,bar">made-up-
scheme://foo,bar</a></p>
```

Example 600

```
<https://../>
```

```
<p><a·
href="https://../">https://../</a>
</p>
```

Example 601

```
<localhost:5001/foo>
```

```
<p><a·
href="localhost:5001/foo">localhost:5
001/foo</a></p>
```

Spaces are not allowed in autolinks:

Example 602

```
<https://foo.bar/baz·bim>
```

```
<p>&lt;https://foo.bar/baz·bim&gt;
</p>
```

Backslash-escapes do not work inside autolinks:

Example 603

```
<https://example.com/\[\>
```

```
<p><a·
href="https://example.com/%5C%5B%5C">
https://example.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 604

```
<foo@bar.example.com>
```

```
<p><a·
href="mailto:foo@bar.example.com">foo
@bar.example.com</a></p>
```

Example 605

```
<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 606

```
<foo\+@bar.example.com>                 <p>&lt;foo+@bar.example.com&gt;</p>
```

These are not autolinks:

Example 607

```
<>                                      <p>&lt;&gt;</p>
```

Example 608

```
<·https://foo.bar·>                     <p>&lt;·https://foo.bar·&gt;</p>
```

Example 609

```
<m:abc>                                 <p>&lt;m:abc&gt;</p>
```

Example 610

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

Example 611

```
https://example.com                     <p>https://example.com</p>
```

Example 612

```
foo@bar.example.com                     <p>foo@bar.example.com</p>
```

6.6 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 spaces, tabs, and up to one line ending, 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 spaces, tabs, and up to one line ending, a `=` character, optional spaces, tabs, and up to one line ending, 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 spaces, tabs, line endings, `"`, `'`, `=`, `<`, `>`, 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 spaces, tabs, and up to one line ending, an optional `/` character, and a `>` character.

A **closing tag** consists of the string `</`, a tag name, optional spaces, tabs, and up to one line ending, and the character `>`.

An **HTML comment** consists of `<!-->`, `<!--->`, or `<!--`, a string of characters not including the string `-->`, and `-->` (see the HTML 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 `<!`, an ASCII letter, zero or more 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 613

```
<a><bab><c2c>
```

```
<p><a><bab><c2c></p>
```

Empty elements:

Example 614

```
<a/><b2/>
```

```
<p><a/><b2/></p>
```

Whitespace is allowed:

Example 615

```
<a→→/><b2
data="foo"→>
```

```
<p><a→→/><b2
data="foo"→></p>
```

With attributes:

Example 616

```
<a·foo="bar"·bam·=·'baz·<em>"</em>'
_boolean·zoop:33=zoop:33·/>
```

```
<p><a·foo="bar"·bam·=·'baz·<em>"</em>'
_boolean·zoop:33=zoop:33·/></p>
```

Custom tag names can be used:

Example 617

```
Foo·<responsive-image·src="foo.jpg"·/>
```

```
<p>Foo·<responsive-image·src="foo.jpg"·/></p>
```

Illegal tag names, not parsed as HTML:

Example 618

```
<33>·<__>
```

```
<p>&lt;33&gt;·&lt;__&gt;</p>
```

Illegal attribute names:

Example 619

```
<a·h*#ref="hi">
```

```
<p>&lt;a·h*#ref=&quot;hi&quot;&gt;</p>
```

Illegal attribute values:

Example 620

```
<a·href="hi'>·<a·href=hi'>
```

```
<p>&lt;a·href=&quot;hi'&gt;·&lt;a·href=hi'&gt;</p>
```

Illegal whitespace:

Example 621

```
<·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 622

```
<a·href='bar'title=title>
```

```
<p>&lt;a·href='bar'title=title&gt;
</p>
```

Closing tags:

Example 623

```
</a></foo·>
```

```
<p></a></foo·></p>
```

Illegal attributes in closing tag:

Example 624

```
</a·href="foo">
```

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

Comments:

Example 625

```
foo·<!--·this·is·a·--
comment·-·with·hyphens·-->
```

```
<p>foo·<!--·this·is·a·--
comment·-·with·hyphens·--></p>
```

Example 626

```
foo·<!-->·foo·-->

foo·<!--->·foo·-->
```

```
<p>foo·<!-->·foo·--&gt;</p>
<p>foo·<!--->·foo·--&gt;</p>
```

Processing instructions:

Example 627

```
foo·<?php·echo·$a;·?>
```

```
<p>foo·<?php·echo·$a;·?></p>
```

Declarations:

Example 628

```
foo·<!ELEMENT·br·EMPTY>
```

```
<p>foo·<!ELEMENT·br·EMPTY></p>
```

CDATA sections:

Example 629

```
foo·<![CDATA[>&<]]>
```

```
<p>foo·<![CDATA[>&<]]></p>
```

Entity and numeric character references are preserved in HTML attributes:

Example 630

```
foo·<a·href="&ouml;">
```

```
<p>foo·<a·href="&ouml;"></p>
```

Backslash escapes do not work in HTML attributes:

Example 631

```
foo·<a·href="\*">
```

```
<p>foo·<a·href="\*"></p>
```

Example 632

```
<a·href="\"">
```

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

6.7 Hard line breaks

A line ending (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 633

```
foo··
baz
```

```
<p>foo<br·/>
baz</p>
```

For a more visible alternative, a backslash before the line ending may be used instead of two or more spaces:

Example 634

```
foo\
baz
```

```
<p>foo<br·/>
baz</p>
```

More than two spaces can be used:

Example 635

```
foo·······
baz
```

```
<p>foo<br·/>
baz</p>
```

Leading spaces at the beginning of the next line are ignored:

Example 636

```
foo··
·····bar
```

```
<p>foo<br·/>
bar</p>
```

Example 637

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

Hard line breaks can occur inside emphasis, links, and other constructs that allow inline content:

Example 638

```
*foo··                                      <p><em>foo<br·/>
bar*                                        bar</em></p>
```

Example 639

```
*foo\                                       <p><em>foo<br·/>
bar*                                        bar</em></p>
```

Hard line breaks do not occur inside code spans

Example 640

```
`code··                                     <p><code>code···span</code></p>
span`
```

Example 641

```
`code\                                      <p><code>code\·span</code></p>
span`
```

or HTML tags:

Example 642

```
<a·href="foo··                              <p><a·href="foo··
bar">                                       bar"></p>
```

Example 643

```
<a·href="foo\                               <p><a·href="foo\
bar">                                       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 644

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

Example 645

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

Example 646

```
###·foo\                                <h3>foo\</h3>
```

Example 647

```
###·foo··                               <h3>foo</h3>
```

6.8 Soft line breaks

A regular line ending (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 soft line break 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 648

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

Spaces at the end of the line and beginning of the next line are removed:

Example 649

```
foo·                                    <p>foo
·baz                                    baz</p>
```

A conforming parser may render a soft line break in HTML either as a line ending or as a space.

A renderer may also provide an option to render soft line breaks as hard line breaks.

6.9 Textual content

Any characters not given an interpretation by the above rules will be parsed as plain textual content.

Example 650

```
hello·$.;'there                         <p>hello·$.;'there</p>
```

Example 651

```
Foo·χρῆν                                <p>Foo·χρῆν</p>
```

Internal spaces are preserved verbatim:

Example 652

```
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 container 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, collapsed 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 (`*`, `_`), indexed to the length of the closing delimiter run (modulo 3) and to whether the closing delimiter can also be an opener. 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 >= 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.