GitHub Flavored Markdown Spec

github.github.com/gfm

GitHub Flavored Markdown, often shortened as GFM, is the dialect of Markdown that is currently supported for user content on GitHub.com and GitHub Enterprise.

This formal specification, based on the CommonMark Spec, defines the syntax and semantics of this dialect.

GFM is a strict superset of CommonMark. All the features which are supported in GitHub user content and that are not specified on the original CommonMark Spec are hence known as **extensions**, and highlighted as such.

While GFM supports a wide range of inputs, it's worth noting that GitHub.com and GitHub Enterprise perform additional post-processing and sanitization after GFM is converted to HTML to ensure security and consistency of the website.

Markdown is a plain text format for writing structured documents, based on conventions for indicating formatting in email and usenet posts. It was developed by John Gruber (with help from Aaron Swartz) and released in 2004 in the form of a syntax description and a Perl script (`Markdown.pl`) for converting Markdown to HTML. In the next decade, dozens of implementations were developed in many languages. Some extended the original Markdown syntax with conventions for footnotes, tables, and other document elements. Some allowed Markdown documents to be rendered in formats other than HTML. Websites like Reddit, StackOverflow, and GitHub had millions of people using Markdown. And Markdown started to be used beyond the web, to author books, articles, slide shows, letters, and lecture notes.

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

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

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

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.

- 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.)

- 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.)

- 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?
  ```

- 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.)

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

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

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

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

- 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
  ```

- 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).
  ```

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

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

- 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`
  ```

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

  ```
  - # Heading
  ```

- Can list items be empty?

  ```
  * a
  *
  * b
  ```

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

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

- 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.

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:

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

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

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

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

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

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

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

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

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

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

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 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 9

```
 - foo
   - bar
→ - baz
```

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

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

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.

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:

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.

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

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

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

Example 25

```
_ _ _ _ a

a------

---a---
```

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

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

Example 27

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

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

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:

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

Example 30

```
* Foo
* * *
* Bar
```

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

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

Example 32

```
# foo
## foo
### foo
#### foo
##### foo
###### foo
```

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

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

Example 42

```
# foo ##################################
##### foo ##
```

```
<h1>foo</h1>
<h5>foo</h5>
```

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

Example 46

```
### foo \###
## foo #\##
# foo \#
```

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

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

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

The heading is a level 1 heading if `=` characters are used in the setext heading underline, and a level 2 heading if `-` characters are used. The contents of the heading are the result of parsing the preceding lines of text as CommonMark inline content.

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

Example 50

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

Foo *bar*
---------

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

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

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

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

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

Example 61

```
`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:

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

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:

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

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

- paragraph “Foo”, heading “bar”, paragraph “baz”
- paragraph “Foo bar”, thematic break, paragraph “baz”
- paragraph “Foo bar — baz”
- 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:

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

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

Example 77

```
    a simple
      indented code block

<pre><code>a simple
  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:

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

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

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

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

Example 85

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

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

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

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

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

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

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

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

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

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

Example 98

````
> ```
> aaa

bbb
````

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

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

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

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

Example 110

~~~
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 111

```
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 112

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

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

Example 113

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

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

Example 116

```
~~~ aa ``` ~~~
foo
~~~
```

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

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

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

- **Start condition:** line begins with the string `<script`, `<pre`, or `<style` (case-insensitive), followed by whitespace, the string `>`, or the end of the line.
  End condition: line contains an end tag `</script>`, `</pre>`, or `</style>` (case-insensitive; it need not match the start tag).

- **Start condition:** line begins with the string `<!--`.
  End condition: line contains the string `-->`.

- **Start condition:** line begins with the string `<?`.
  End condition: line contains the string `?>`.

- **Start condition:** line begins with the string `<!` followed by an uppercase ASCII letter.
  End condition: line contains the character `>`.

- **Start condition:** line begins with the string `<![CDATA[`.
  End condition: line contains the string `]]>`.

- **Start condition:** line begins with a complete open tag (with any tag name other than `script`, `style`, or `pre`) or a complete closing tag, followed only by whitespace or the end of the line.
  End condition: line is followed by a blank line.

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

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

Example 118

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

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

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

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

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

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

Example 119

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

okay.
```

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

Example 122

```
<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 124

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

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

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

Example 129

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

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

Example 130

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

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

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

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):

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:

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:

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.)

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

Example 139

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

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

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

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

Example 140

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

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

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

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

Example 141

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

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

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

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

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

Example 145

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

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

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

Example 151

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

  } else {

    return 0;
  }
}
]]>
okay
```

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

  } else {

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

Example 152

```
  <!-- foo -->

    <!-- foo -->
```

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

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

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

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

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

- 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:

Example 157

```
<div>

*Emphasized* text.

</div>
```

```
<div>
<p><em>Emphasized</em> text.</p>
</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 159

```
<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 160

```
<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.

Example 161

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

[foo]
```

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

Example 162

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

[foo]
```

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

Example 163

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

[Foo*bar\]]
```

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

Example 164

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

[Foo bar]
```

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

Example 165

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

[foo]
```

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

Example 166

```
[foo]: /url 'title

with blank line'

[foo]
```

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

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

Example 171

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

[foo]
```

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

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

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

This is not a link reference definition, because there are non-whitespace characters after the title:

Example 180

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

[foo]
```

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

Example 181

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

[foo]
~~~

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

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

Example 183

```
# [Foo]
[foo]: /url
> bar
```

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

Example 186

```
[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 187

```
[foo]

> [foo]: /url
```

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

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

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

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

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

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

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

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.)

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

-
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*.

- **Laziness.** If a string of lines *Ls* constitute a block quote with contents *Bs*, then the result of deleting the initial block quote marker from one or more lines in which the next non-whitespace character after the block quote marker is paragraph continuation text is a block quote with *Bs* as its content. Paragraph continuation text is text that will be parsed as part of the content of a paragraph, but does not occur at the beginning of the paragraph.

- **Consecutiveness.** A document cannot contain two block quotes in a row unless there is a blank line between them.

Example 206

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

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

Example 207

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

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

Example 208

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

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

Example 209

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

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

Example 210

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

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

Example 213

```
> - 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 214

```
>     foo
    bar
```

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

Example 215

~~~
> ```
foo
```
~~~

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

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.

Example 220

```
> 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 224

```
> 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:

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 228

```
> > > foo
bar
```

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

Example 229

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

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

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

Example 230

```
>     code

>    not code
```

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

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.)

-

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

Exceptions:

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

Example 231

```
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 232

```
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, and there are three spaces between the list marker and the next non-whitespace character, then blocks must be indented five spaces in order to fall under the list item.

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

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

Example 237

```
   > > 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 238

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

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

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

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

Example 241

~~~
1.  foo

    ```
    bar
    ```

    baz

    > bam
~~~

```
<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 242

```
- Foo

      bar


      baz
```

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


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

- **Item starting with indented code.** If a sequence of lines *Ls* constitute a sequence of blocks *Bs* starting with an indented code block, and *M* is a list marker of width *W* followed by one space, then the result of prepending *M* and the following space to the first line of *Ls*, and indenting subsequent lines of *Ls* by *W + 1* spaces, is a list item with *Bs* as its contents. If a line is empty, then it need not be indented. The type of the list item (bullet or ordered) is determined by the type of its list marker. If the list item is ordered, then it is also assigned a start number, based on the ordered list marker.
An indented code block will have to be indented four spaces beyond the edge of the region where text will be included in the list item. In the following case that is 6 spaces:

Example 248

```
- foo

      bar
```

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

Example 249

```
  10.  foo

           bar
```

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

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

Example 250

```
    indented code

paragraph

    more code
```

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

Example 251

```
1.     indented code

   paragraph

       more code
```

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

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

Example 252

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

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

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

Example 256

```
-
  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:

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

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

Example 264

```
 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>
```

Example 265

```
  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>
```

Example 266

```
   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>
```

Example 267

```
    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>
```

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

Example 268

```
  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>
```

Example 269

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

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

Example 270

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

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

Example 271

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

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

- **That's all.** Nothing that is not counted as a list item by rules #1–5 counts as a list item.

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

Example 272

```
- foo
  - bar
    - baz
      - boo
```

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

Example 273

```
- foo
 - bar
  - baz
   - boo
```

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

Example 274

```
10) foo
    - bar
```

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

Example 275

```
10) foo
   - bar
```

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

Example 277

```
1. - 2. foo
```

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

Example 278

```
- # Foo
- Bar
  ---
  baz
```

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

- “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.”

- “To make lists look nice, you can wrap items with hanging indents.... But if you don’t want to, you don’t have to.”

- “List items may consist of multiple paragraphs. Each subsequent paragraph in a list item must be indented by either 4 spaces or one tab.”

- “It looks nice if you indent every line of the subsequent paragraphs, but here again, Markdown will allow you to be lazy.”

- “To put a blockquote within a list item, the blockquote’s `>` delimiters need to be indented.”

- “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

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

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

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` :

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:

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.

5.3Task list items (extension)

GFM enables the `tasklist` extension, where an additional processing step is performed on list items.

A task list item is a list item where the first block in it is a paragraph which begins with a task list item marker and at least one whitespace character before any other content.

A task list item marker consists of an optional number of spaces, a left bracket (`[`), either a whitespace character or the letter `x` in either lowercase or uppercase, and then a right bracket (`]`).

When rendered, the task list item marker is replaced with a semantic checkbox element; in an HTML output, this would be an `<input type="checkbox">` element.

If the character between the brackets is a whitespace character, the checkbox is unchecked. Otherwise, the checkbox is checked.

This spec does not define how the checkbox elements are interacted with: in practice, implementors are free to render the checkboxes as disabled or inmutable elements, or they may dynamically handle dynamic interactions (i.e. checking, unchecking) in the final rendered document.

Example 279

```
- [ ] foo
- [x] bar
```

```
<ul>
<li><input disabled="" type="checkbox"> foo</li>
<li><input checked="" disabled="" type="checkbox"> bar</li>
</ul>
```

Task lists can be arbitrarily nested:

Example 280

```
- [x] foo
  - [ ] bar
  - [x] baz
- [ ] bim
```

```
<ul>
<li><input checked="" disabled="" type="checkbox"> foo
<ul>
<li><input disabled="" type="checkbox"> bar</li>
<li><input checked="" disabled="" type="checkbox"> baz</li>
</ul>
</li>
<li><input disabled="" type="checkbox"> bim</li>
</ul>
```

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 `)`).

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.)

Example 281

```
- foo
- bar
+ baz
```

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

Example 282

```
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:

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

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:

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

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

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

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

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

Example 284

```
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>
```

Example 285

```
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.

Example 286

```
- foo

- bar


- baz
```

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

Example 287

```
- 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 288

```
- foo
- bar

<!-- -->

- baz
- bim
```

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

Example 289

```
-   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 290

```
- 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 291

```
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 indented more than three spaces. Here `- e` is treated as a paragraph continuation line, because it is indented more than three spaces:

Example 292

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

```
<ul>
<li>a</li>
<li>b</li>
<li>c</li>
<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 293

```
1. a

  2. b

    3. c
```

```
<ol>
<li>
<p>a</p>
</li>
<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 294

```
- a
- b

- c
```

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

Example 295

```
* a
*

* c
```

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

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

Example 296

```
- 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 297

```
- a
- b

  [ref]: /url
- d
```

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

Example 298

~~~
- 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 299

```
- a
  - b

    c
- d
```

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

Example 300

```
* 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 301

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

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

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

Example 304

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

   bar
~~~

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

Example 305

```
* foo
  * bar

  baz
```

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

Example 306

```
- a
  - b
  - c

- d
  - e
  - f
```

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

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

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

Example 308

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

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

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

Example 310

```
\*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>
```

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

Example 316

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

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

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

Example 318

```
[foo](/bar\* "ti\*tle")
```

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

Example 319

```
[foo]

[foo]: /bar\* "ti\*tle"
```

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

Example 320

~~~
``` foo\+bar
foo
```
~~~

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

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.

Example 321

```
&nbsp; &amp; &copy; &AElig; &Dcaron;
&frac34; &HilbertSpace; &DifferentialD;
&ClockwiseContourIntegral; &ngE;
```

```
<p>   &amp; © Æ Ď
¾ ℋ ⅆ
∲ ≧̸</p>
```

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`.

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 324

```
&nbsp &x; &#; &#x;
&#87654321;
&#abcdef0;
&ThisIsNotDefined; &hi?;
```

```
<p>&amp;nbsp &amp;x; &amp;#; &amp;#x;
&amp;#87654321;
&amp;#abcdef0;
&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:

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

Example 328

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

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

Example 329

```
[foo]

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

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

Example 330

~~~
``` 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:

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

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

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

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

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

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

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

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.

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:

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

Example 355

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

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

Example 356

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

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

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

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

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:

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

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.)

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.

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.)

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

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

- An interpretation `<em><strong>...</strong></em>` is always preferred to `<strong><em>...</em></strong>`.

- 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>`.

- 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>`.

- 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.

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

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:

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

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

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

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

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:

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

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

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

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:

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

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

Example 398

```
__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:

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

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

Example 403

```
**Gomphocarpus (*Gomphocarpus physocarpus*, syn.
*Asclepias physocarpa*)**
```

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

Example 404

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

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

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

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

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

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

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:

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

When the lengths of the interior closing and opening delimiter runs are *both* multiples of 3, though, they can match to create emphasis:

Example 426

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

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

Example 427

```
*foo **bar *baz* bim** bop*
```

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

Example 428

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

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

Example 430

```
**** is not an empty strong emphasis
```

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

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

Example 431

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

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

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

Example 434

```
__foo __bar__ baz__
```

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

Example 441

```
**foo *bar **baz**
bim* bop**
```

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

Example 442

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

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

Example 444

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

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

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:

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

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

Example 475

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

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

Example 479

```
*foo __bar *baz bim__ bam*
```

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

Example 489

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

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

Example 490

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

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

6.5Strikethrough (extension)

GFM enables the `strikethrough` extension, where an additional emphasis type is available.

Strikethrough text is any text wrapped in two tildes (`~`).

Example 491

```
~~Hi~~ Hello, world!
```

```
<p><del>Hi</del> Hello, world!</p>
```

As with regular emphasis delimiters, a new paragraph will cause strikethrough parsing to cease:

Example 492

```
This ~~has a

new paragraph~~.
```

```
<p>This ~~has a</p>
<p>new paragraph~~.</p>
```

A link contains link text (the visible text), a link destination (the URI that is the link destination), and optionally a link title. There are two basic kinds of links in Markdown. In inline links the destination and title are given immediately after the link text. In reference links the destination and title are defined elsewhere in the document.

A link text consists of a sequence of zero or more inline elements enclosed by square brackets (`[` and `]`). The following rules apply:

- Links may not contain other links, at any level of nesting. If multiple otherwise valid link definitions appear nested inside each other, the inner-most definition is used.

- Brackets are allowed in the link text only if (a) they are backslash-escaped or (b) they appear as a matched pair of brackets, with an open bracket `[`, a sequence of zero or more inlines, and a close bracket `]`.

- Backtick code spans, autolinks, and raw HTML tags bind more tightly than the brackets in link text. Thus, for example, ``[foo`]` `` could not be a link text, since the second `]` is part of a code span.

- The brackets in link text bind more tightly than markers for emphasis and strong emphasis. Thus, for example, `*[foo*](url)` is a link.

- a sequence of zero or more characters between an opening `<` and a closing `>` that contains no line breaks or unescaped `<` or `>` characters, or

- a nonempty sequence of characters that does not start with `<`, does not include ASCII space or control characters, and includes parentheses only if (a) they are backslash-escaped or (b) they are part of a balanced pair of unescaped parentheses. (Implementations may impose limits on parentheses nesting to avoid performance issues, but at least three levels of nesting should be supported.)

- a 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.

An inline link consists of a link text followed immediately by a left parenthesis `(`, optional whitespace, an optional link destination, an optional link title separated from the link destination by whitespace, optional whitespace, and a right parenthesis `)`. The link's text consists of the inlines contained in the link text (excluding the enclosing square brackets). The link's URI consists of the link destination, excluding enclosing `<...>` if present, with backslash-escapes in effect as described above. The link's title consists of the link title, excluding its enclosing delimiters, with backslash-escapes in effect as described above.

Example 503

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

```
<p>[a](&lt;b)c
[a](&lt;b)c&gt;
[a](<b>c)</p>
```

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

However, if you have unbalanced parentheses, you need to escape or use the `<...>` form:

Example 509

```
[link](#fragment)

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

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

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

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 513

```
[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 514

```
[link](/url "title \"&quot;")
```

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

Titles must be separated from the link using a whitespace. Other Unicode whitespace like non-breaking space doesn't work.

Example 516

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

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

Example 517

```
[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.)

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

Example 524

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

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

Example 525

```
[![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 527

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

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

Example 528

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

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

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

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

Example 534

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

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

A link label begins with a left bracket (`[`) and ends with the first right bracket (`]`) that is not backslash-escaped. Between these brackets there must be at least one non-whitespace character. Unescaped square bracket characters are not allowed inside the opening and closing square brackets of link labels. A link label can have at most 999 characters inside the square brackets.

One label matches another just in case their normalized forms are equal. To normalize a label, strip off the opening and closing brackets, perform the *Unicode case fold*, strip leading and trailing whitespace and collapse consecutive internal whitespace to a single space. If there are multiple matching reference link definitions, the one that comes first in the document is used. (It is desirable in such cases to emit a warning.)

Example 535

```
[foo][bar]

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

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

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

Example 536

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

[ref]: /uri
```

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

Example 538

```
[link *foo **bar** `#`*][ref]

[ref]: /uri
```

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

Example 539

```
[![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 540

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

[ref]: /uri
```

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

Example 541

```
[foo *bar [baz][ref]*][ref]

[ref]: /uri
```

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

The following cases illustrate the precedence of link text grouping over emphasis grouping:

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

Example 546

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

[ref]: /uri
```

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

Example 547

```
[foo][BaR]

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

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

Consecutive internal whitespace is treated as one space for purposes of determining matching:

Example 550

```
[foo] [bar]

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

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

Example 551

```
[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:

(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.)

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 555

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

[ref[bar]]: /uri
```

```
<p>[foo][ref[bar]]</p>
<p>[ref[bar]]: /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 first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching reference link definition. Thus, `[foo][]` is equivalent to `[foo][foo]`.

Example 561

```
[foo][]

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

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

Example 562

```
[*foo* bar][]

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

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

Example 563

```
[Foo][]

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

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

As with full reference links, whitespace is not allowed between the two sets of brackets:

Example 564

```
[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 first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching link reference definition. Thus, `[foo]` is equivalent to `[foo][]`.

Example 565

```
[foo]

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

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

Example 566

```
[*foo* bar]

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

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

Example 567

```
[[*foo* bar]]

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

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

Example 569

```
[Foo]

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

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

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

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

Example 576

```
[foo](not a link)

[foo]: /url1
```

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

Example 578

```
[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 579

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

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

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

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 580

```
![foo](/url "title")
```

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

Example 581

```
![foo *bar*]

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

```
<p><img src="train.jpg" alt="foo bar" title="train &amp; tracks" /></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 584

```
![foo *bar*][]

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

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

Example 585

```
![foo *bar*][foobar]

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

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

Example 587

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

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

Example 592

```
![foo][]

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

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

Example 593

```
![*foo* bar][]

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

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

Example 594

```
![Foo][]

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

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

As with reference links, whitespace is not allowed between the two sets of brackets:

Example 595

```
![foo]
[]

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

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

Example 596

```
![foo]

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

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

Example 597

```
![*foo* bar]

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

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

Example 598

```
![[foo]]

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

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

Example 599

```
![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 601

```
\![foo]

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

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

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.

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 ("-").

Example 602

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

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

Example 603

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

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

Example 604

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

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

Example 605

```
<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 607

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

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

Example 609

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

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

Example 611

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

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

Example 612

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

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

Example 613

```
<foo+special@Bar.baz-bar0.com>
```

```
<p><a href="mailto:foo+special@Bar.baz-bar0.com">foo+special@Bar.baz-bar0.com</a>
</p>
```

6.9Autolinks (extension)

GFM enables the `autolink` extension, where autolinks will be recognised in a greater number of conditions.

Autolinks can also be constructed without requiring the use of `<` and to `>` to delimit them, although they will be recognized under a smaller set of circumstances. All such recognized autolinks can only come at the beginning of a line, after whitespace, or any of the delimiting characters `*`, `_`, `~`, and `(`.

An extended www autolink will be recognized when the text `www.` is found followed by a valid domain. A valid domain consists of segments of alphanumeric characters, underscores (`_`) and hyphens (`-`) separated by periods (`.`). There must be at least one period, and no underscores may be present in the last two segments of the domain.

The scheme `http` will be inserted automatically:

Example 621

```
www.commonmark.org
```

```
<p><a href="http://www.commonmark.org">www.commonmark.org</a></p>
```

After a valid domain, zero or more non-space non-`<` characters may follow:

Example 622

```
Visit www.commonmark.org/help for more information.
```

```
<p>Visit <a href="http://www.commonmark.org/help">www.commonmark.org/help</a> for
more information.</p>
```

We then apply extended autolink path validation as follows:

Trailing punctuation (specifically, `?`, `!`, `.`, `,`, `:`, `*`, `_`, and `~`) will not be considered part of the autolink, though they may be included in the interior of the link:

Example 623

```
Visit www.commonmark.org.

Visit www.commonmark.org/a.b.
```

```
<p>Visit <a href="http://www.commonmark.org">www.commonmark.org</a>.</p>
<p>Visit <a href="http://www.commonmark.org/a.b">www.commonmark.org/a.b</a>.</p>
```

When an autolink ends in `)`, we scan the entire autolink for the total number of parentheses. If there is a greater number of closing parentheses than opening ones, we don't consider the unmatched trailing parentheses part of the autolink, in order to facilitate including an autolink inside a parenthesis:

Example 624

```
www.google.com/search?q=Markup+(business)

www.google.com/search?q=Markup+(business)))

(www.google.com/search?q=Markup+(business))

(www.google.com/search?q=Markup+(business)
```

```
<p><a href="http://www.google.com/search?q=Markup+
(business)">www.google.com/search?q=Markup+(business)</a></p>
<p><a href="http://www.google.com/search?q=Markup+
(business)">www.google.com/search?q=Markup+(business)</a>))</p>
<p>(<a href="http://www.google.com/search?q=Markup+
(business)">www.google.com/search?q=Markup+(business)</a>)</p>
<p>(<a href="http://www.google.com/search?q=Markup+
(business)">www.google.com/search?q=Markup+(business)</a></p>
```

This check is only done when the link ends in a closing parentheses `)`, so if the only parentheses are in the interior of the autolink, no special rules are applied:

Example 625

```
www.google.com/search?q=(business))+ok
```

```
<p><a href="http://www.google.com/search?q=(business))+ok">www.google.com/search?
q=(business))+ok</a></p>
```

If an autolink ends in a semicolon (`;`), we check to see if it appears to resemble an entity reference; if the preceding text is `&` followed by one or more alphanumeric characters. If so, it is excluded from the autolink:

Example 626

```
www.google.com/search?q=commonmark&hl=en

www.google.com/search?q=commonmark&hl;
```

```
<p><a href="http://www.google.com/search?
q=commonmark&amp;hl=en">www.google.com/search?q=commonmark&amp;hl=en</a></p>
<p><a href="http://www.google.com/search?q=commonmark">www.google.com/search?
q=commonmark</a>&amp;hl;</p>
```

`<` immediately ends an autolink.

Example 627

```
www.commonmark.org/he<lp
```

```
<p><a href="http://www.commonmark.org/he">www.commonmark.org/he</a>&lt;lp</p>
```

An extended url autolink will be recognised when one of the schemes `http://`, or `https://`, followed by a valid domain, then zero or more non-space non-`<` characters according to extended autolink path validation:

Example 628

```
http://commonmark.org

(Visit https://encrypted.google.com/search?q=Markup+(business))
```

```
<p><a href="http://commonmark.org">http://commonmark.org</a></p>
<p>(Visit <a href="https://encrypted.google.com/search?q=Markup+
(business)">https://encrypted.google.com/search?q=Markup+(business)</a>)</p>
```

An extended email autolink will be recognised when an email address is recognised within any text node. Email addresses are recognised according to the following rules:

- One ore more characters which are alphanumeric, or `.`, `-`, `_`, or `+`.
- An `@` symbol.
- One or more characters which are alphanumeric, or `-` or `_`, separated by periods (`.`). There must be at least one period. The last character must not be one of `-` or `_`.

The scheme `mailto:` will automatically be added to the generated link:

Example 629

```
foo@bar.baz
```

```
<p><a href="mailto:foo@bar.baz">foo@bar.baz</a></p>
```

`+` can occur before the `@`, but not after.

Example 630

```
hello@mail+xyz.example isn't valid, but hello+xyz@mail.example is.
```

```
<p>hello@mail+xyz.example isn't valid, but <a
href="mailto:hello+xyz@mail.example">hello+xyz@mail.example</a> is.</p>
```

`.`, `-`, and `_` can occur on both sides of the `@`, but only `.` may occur at the end of the email address, in which case it will not be considered part of the address:

Example 631

```
a.b-c_d@a.b

a.b-c_d@a.b.

a.b-c_d@a.b-

a.b-c_d@a.b_
```

```
<p><a href="mailto:a.b-c_d@a.b">a.b-c_d@a.b</a></p>
<p><a href="mailto:a.b-c_d@a.b">a.b-c_d@a.b</a>.</p>
<p>a.b-c_d@a.b-</p>
<p>a.b-c_d@a.b_</p>
```

- **Start condition:** line begins the string `<` or `</` followed by one of the strings (case-insensitive) `address`, `article`, `aside`, `base`, `basefont`, `blockquote`, `body`, `caption`, `center`, `col`, `colgroup`, `dd`, `details`, `dialog`, `dir`, `div`, `dl`, `dt`, `fieldset`, `figcaption`, `figure`, `footer`, `form`, `frame`, `frameset`, `h1`, `h2`, `h3`, `h4`, `h5`, `h6`, `head`, `header`, `hr`, `html`, `iframe`, `legend`, `li`, `link`, `main`, `menu`, `menuitem`, `nav`, `noframes`, `ol`, `optgroup`, `option`, `p`, `param`, `section`, `source`, `summary`, `table`, `tbody`, `td`, `tfoot`, `th`, `thead`, `title`, `tr`, `track`, `ul`, followed by whitespace, the end of the line, the string `>`, or the string `/>`.
End condition: line is followed by a blank line.

4.10Tables (extension)

GFM enables the `table` extension, where an additional leaf block type is available.

A table is an arrangement of data with rows and columns, consisting of a single header row, a delimiter row separating the header from the data, and zero or more data rows.

Each row consists of cells containing arbitrary text, in which inlines are parsed, separated by pipes (`|`). A leading and trailing pipe is also recommended for clarity of reading, and if there's otherwise parsing ambiguity. Spaces between pipes and cell content are trimmed. Block-level elements cannot be inserted in a table.

The delimiter row consists of cells whose only content are hyphens (`-`), and optionally, a leading or trailing colon (`:`), or both, to indicate left, right, or center alignment respectively.

Example 198

```
| foo | bar |
| --- | --- |
| baz | bim |
<table>
<thead>
<tr>
<th>foo</th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>baz</td>
<td>bim</td>
</tr>
</tbody>
</table>
```

Cells in one column don’t need to match length, though it’s easier to read if they are. Likewise, use of leading and trailing pipes may be inconsistent:

Example 199

```
| abc | defghi |
:-: | -----------:
bar | baz
```

```
<table>
<thead>
<tr>
<th align="center">abc</th>
<th align="right">defghi</th>
</tr>
</thead>
<tbody>
<tr>
<td align="center">bar</td>
<td align="right">baz</td>
</tr>
</tbody>
</table>
```

Include a pipe in a cell’s content by escaping it, including inside other inline spans:

Example 200

```
| f\|oo  |
| ------ |
| b `\|` az |
| b **\|** im |
<table>
<thead>
<tr>
<th>f|oo</th>
</tr>
</thead>
<tbody>
<tr>
<td>b <code>|</code> az</td>
</tr>
<tr>
<td>b <strong>|</strong> im</td>
</tr>
</tbody>
</table>
```

The table is broken at the first empty line, or beginning of another block-level structure:

Example 201

```
| abc | def |
| --- | --- |
| bar | baz |
> bar
```

```
<table>
<thead>
<tr>
<th>abc</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>baz</td>
</tr>
</tbody>
</table>
<blockquote>
<p>bar</p>
</blockquote>
```

Example 202

```
| abc | def |
| --- | --- |
| bar | baz |
bar

bar
```

```
<table>
<thead>
<tr>
<th>abc</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>baz</td>
</tr>
<tr>
<td>bar</td>
<td></td>
</tr>
</tbody>
</table>
<p>bar</p>
```

The header row must match the delimiter row in the number of cells. If not, a table will not be recognized:

Example 203

```
| abc | def |
| --- |
| bar |
```

```
<p>| abc | def |
| --- |
| bar |</p>
```

The remainder of the table’s rows may vary in the number of cells. If there are a number of cells fewer than the number of cells in the header row, empty cells are inserted. If there are greater, the excess is ignored:

Example 204

```
| abc | def |
| --- | --- |
| bar |
| bar | baz | boo |
<table>
<thead>
<tr>
<th>abc</th>
<th>def</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td></td>
</tr>
<tr>
<td>bar</td>
<td>baz</td>
</tr>
</tbody>
</table>
```

If there are no rows in the body, no `<tbody>` is generated in HTML output:

Example 205

```
| abc | def |
| --- | --- |
```

```
<table>
<thead>
<tr>
<th>abc</th>
<th>def</th>
</tr>
</thead>
</table>
```

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.

A tag name consists of an ASCII letter followed by zero or more ASCII letters, digits, or hyphens (`-`).

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 HTML comment consists of `<!--` + *text* + `-->` , where *text* does not start with `>` or `->` , does not end with `-` , and does not contain `--` . (See the HTML5 spec.)

A processing instruction consists of the string `<?` , a string of characters not including the string `?>` , and the string `?>` .

A declaration consists of the string `<!` , a name consisting of one or more uppercase ASCII letters, whitespace, a string of characters not including the character `>` , and the character `>` .

A CDATA section consists of the string `<![CDATA[`, a string of characters not including the string `]]>`, and the string `]]>`.

Example 635

```
<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>
```

Example 636

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

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

Example 639

```
<a href="hi'> <a href=hi'>
```

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

Example 640

```
< 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>
```

Example 644

```
foo <!-- this is a
comment - with hyphen -->
```

```
<p>foo <!-- this is a
comment - with hyphen --></p>
```

Example 645

```
foo <!-- not a comment -- two hyphens -->
```

```
<p>foo &lt;!-- not a comment -- two hyphens --&gt;</p>
```

Example 646

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

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

```
<p>foo &lt;!--&gt; foo --&gt;</p>
<p>foo &lt;!-- foo---&gt;</p>
```

6.11Disallowed Raw HTML (extension)

GFM enables the `tagfilter` extension, where the following HTML tags will be filtered when rendering HTML output:

- `<title>`
- `<textarea>`
- `<style>`
- `<xmp>`
- `<iframe>`
- `<noembed>`
- `<noframes>`
- `<script>`
- `<plaintext>`

Filtering is done by replacing the leading `<` with the entity `&lt;`. These tags are chosen in particular as they change how HTML is interpreted in a way unique to them (i.e. nested HTML is interpreted differently), and this is usually undesireable in the context of other rendered Markdown content.

All other HTML tags are left untouched.

Example 653

```
<strong> <title> <style> <em>

<blockquote>
  <xmp> is disallowed.  <XMP> is also disallowed.
</blockquote>
```

```
<p><strong> &lt;title> &lt;style> <em></p>
<blockquote>
  &lt;xmp> is disallowed.  &lt;XMP> is also disallowed.
</blockquote>
```

A line break (not in a code span or HTML tag) that is preceded by two or more spaces and does not occur at the end of a block is parsed as a hard line break (rendered in HTML as a `<br />` tag):

For a more visible alternative, a backslash before the line ending may be used instead of two spaces:

Line breaks can occur inside emphasis, links, and other constructs that allow inline content:

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:

A regular line break (not in a code span or HTML tag) that is not preceded by two or more spaces or a backslash is parsed as a softbreak. (A softbreak may be rendered in HTML either as a line ending or as a space. The result will be the same in browsers. In the examples here, a line ending will be used.)

A conforming parser may render a soft line break in HTML either as a line break or as a space.

A renderer may also provide an option to render soft line breaks as hard line breaks.

Any characters not given an interpretation by the above rules will be parsed as plain textual content.

In this appendix we describe some features of the parsing strategy used in the CommonMark reference implementations.

-

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.

-

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:

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:

- One or more open blocks may be closed.
- One or more new blocks may be created as children of the last open block.
- 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.

-

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.

- Next, after consuming the continuation markers for existing blocks, we look for new block starts (e.g. `>` for a block quote). If we encounter a new block start, we close any blocks unmatched in step 1 before creating the new block as a child of the last matched block.

- 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).

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:

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`:

is a "lazy continuation" of the open `paragraph`, so it gets added to the paragraph's text:

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`:

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:

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.

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`.

By far the trickiest part of inline parsing is handling emphasis, strong emphasis, links, and images. This is done using the following algorithm.

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.

Starting at the top of the delimiter stack, we look backwards through the stack for an opening `[` or `![` delimiter.

- If we don’t find one, we return a literal text node `]` .

- If we do find one, but it’s not *active*, we remove the inactive delimiter from the stack, and return a literal text node `]` .

- If we find one and it’s active, then we parse ahead to see if we have an inline link/image, reference link/image, compact reference link/image, or shortcut reference link/image.

  - If we don’t, then we remove the opening delimiter from the delimiter stack and return a literal text node `]` .

  - If we do, then

    - We return a link or image node whose children are the inlines after the text node pointed to by the opening delimiter.

    - We run *process emphasis* on these inlines, with the `[` opener as `stack_bottom` .

    - We remove the opening delimiter.

    - If we have a link (and not an image), we also set all `[` delimiters before the opening delimiter to *inactive*. (This will prevent us from getting links within links.)

Parameter `stack_bottom` sets a lower bound to how far we descend in the delimiter stack. If it is NULL, we can go all the way to the bottom. Otherwise, we stop before visiting `stack_bottom` .

Let `current_position` point to the element on the delimiter stack just above `stack_bottom` (or the first element if `stack_bottom` is NULL).

We keep track of the `openers_bottom` for each delimiter type (`*` , `_`) and each length of the closing delimiter run (modulo 3). Initialize this to `stack_bottom` .

- 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.