CommonMark

John MacFarlane

2020 年 06 月 15 日

Contents:

This is a translation of CommonMark Spec.

タイトル: CommonMark Spec 著者: John MacFarlane バージョン: 0.29 日付: '2019-04-06' ライセンス: 'CC-BY-SA 4.0' ...

第 1 章

イントロダクション

1.1 What is Markdown?

Markdown は、電子メールおよび usenet への投稿などで使われていた書式の慣習をベースにした、構造化文書を記述するためのテキスト形式です。Markdown は John Gruber によって (Aaron Swartz の手助けを受けながら)、2004 年に 文法定義と HTML への変換スクリプト (`Markdown.pl`) としてリリースされました。その後の 10 年間で、多くのプログラミング言語で Markdown の処理系が数多く開発されました。いくつかの処理系では、オリジナルの Markdown 文法に脚注やテーブルなどの記法が追加されました。 また、いくつかの処理系では、Markdown 文書から HTML 以外の形式への変換に対応しました。 Reddit や StackOverflow、GitHub などのウェブサイトでは、Markdown は数百万のユーザに使われています。そして、Markdown はウェブだけでなく、書籍や記事、スライド、手紙、講義ノートなどのさまざまな用途に使われはじめています。

Markdown と他の多くの軽量マークアップ言語との違いは、読みやすさにあります。この点について、Gruber は次のようにコメントしています:

> Markdown 記法のもっとも重要な設計目標は、ドキュメントをできるかぎり読みやすくすることです。そのため、Markdown 形式のドキュメントはタグや書籍の指定などでマークアップされているように見えなくても、プレーンテキストとしてそのまま公開可能であることを目指しています。 （http://daringfireball.net/projects/markdown/）

この点は AsciiDoc のドキュメントと Markdown のドキュメントを比較することで説明できます。以下は AsciiDoc のマニュアルに掲載されている AsciiDoc のサンプルです:

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

(次のページに続く)

(前のページからの続き)

```
+
List item continued with a third paragraph.

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

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

b. List item b.

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

そして、こちらは同じ内容を Markdown で記述したものです:

```
1.  List item one.

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

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

    List item continued with a third paragraph.

2.  List item two continued with an open block.

    This paragraph is part of the preceding list item.

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

       This paragraph is part of the preceding list item.

    2. List item b.

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

AsciiDoc バージョンは、間違いなく書くのが簡単です。インデントについて気にしなくて済みます。一方、Markdown バージョンはとても読みやすくなっています。多段リストは、変換されたドキュメントだけでなく、ソースのままでも読みやすいです。

1.2 なぜ仕様が必要なのか?

John Gruber の Markdown の文法の標準的な説明は、その文法を明確に規定していません。以下にその疑問点を挙げます。

1. サブリストを作るにはいくつインデントするのか。仕様では段落を継続するには 4 スペース分のインデントが必要だと説明されていますが、サブリストについては明示的ではありません。段落と同様に、サブリストも 4 スペース分のインデントが必要と考えるのが自然ですが、`Markdown.pl` はそのように実装されていません。この問題は、Markdown 処理系によって動作に差異があるため、実際のドキュメントを処理する際にユーザが驚くことになります。(参考 John Gruber のコメント)

2. 引用ブロックや見出しの直前に空行が必要かどうか。多くの処理系では空行を必要としません。しかし、その動作はハードラップされたテキストにおいて予期せぬ結果をもたらし、解析における曖昧さに繋がる可能性があります (ある処理系は見出しを引用ブロックの内側に置くが、別の処理系ではそうではない)。(John Gruber も 空行を必要とすることに賛成しました)

3. インデントコードブロックの直前に空行が必要かどうか。(`Markdown.pl` は空行を必要としますが、仕様には言及がありません。また、いくつかの処理系は空行を必要としません。)

   ```
   paragraph
       code?
   ```

4. リストの項目が <p> タグで囲われるかどうかがどのように決まるのか。リストの一部だけをゆるめて (loose)、一部をきつくする (tight) ことはできるのか。次のようなリストをどのように扱えばよいのか。

   ```
   1. one

   2. two
   3. three
   ```

   また、次のようなリストはどうでしょうか。

   ```
   1.  one
       - a

       - b
   2.  two
   ```

   (参考: John Gruber による関連コメントは こちら)

5. リスト記号はインデントできるのでしょうか。また、番号付きリストの番号は右寄せできるのでしょうか。

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

6. 次の例は二番目のリスト要素に段落区切りをもつリストと解釈されるのでしょうか。それとも段落区切りで分割されたふたつのリストと解釈されるのでしょうか。

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

7. 箇条書きのマーカーがリストの途中で番号からバレットに変わった場合、いくつの箇条書きとして解釈すべきでしょうか。ひとつ？それともふたつ？ (Markdown の文法ではふたつと定義されていますが、Markdown.pl や多くの実装はひとつのリストと解釈します)

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

8. インライン構造のマーカーの優先順位はどうでしょうか。たとえば、以下の例は有効なリンクでしょうか。それともコードスパンが優先されるべきでしょうか。

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

9. 強調および強い強調のマーカーの優先順位はどうでしょうか。たとえば、以下の例はどのように解釈されるべきでしょうか。

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

10. ブロックレベルの構造とインラインレベルの構造における優先順位はどうでしょうか。たとえば、以下の例はどのように解釈されるべきでしょうか。

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

11. 箇条書きに見出しを含めることはできるのでしょうか。 (`Markdown.pl` では含めることはできません。しかし、ブロック引用に見出しを含めることはできます。)

    ```
    - # Heading
    ```

12. 空の箇条書き要素は認められるべきでしょうか。

    ```
    * a
    *
    * b
    ```

13. リンク参照はブロック引用や箇条書きに含めることはできるのでしょうか。

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

14. 同じ参照に対して複数の定義がある場合、どちらが優先されるのでしょうか。

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

[foo][]
```

仕様が存在しないため、初期の実装者はこうしたあいまいさを解決するために`Markdown.pl`を参考にしました。しかし、`Markdown.pl`はとてもバグが多く、また多くの場合に悪い結果をもたらしたため、仕様の代わりにはなりませんでした。

こうして、明確な仕様が存在しないことによって実装は大きく分岐しました。その結果、あるシステム (たとえば GitHub wiki) でレンダリングされたドキュメントと別のシステム (たとえば pandoc で docbook に変換した場合) でレンダリングされたものが異なるのをよく見かけます。残念なことに、Markdown は構文エラーを発生させないため、こうした差異はすぐには発見されませんことがほとんどです。

1.3 この文書について

このドキュメントでは Markdown の文法を明確に規定しようとしています。また、このドキュメントには Markdown の例と HTML への変換結果を並べています。これらは適合テストを兼ねることを目的としています。付属する`spec_test.py`を利用すると、任意の Markdown プログラムに対してテストを実行できます。

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

このドキュメントでは Markdown を抽象構文木に変換する方法を説明しているため、HTML のかわりに構文木の抽象表現を利用するのが理にかなっています。しかし、HTML は構造上の違いを表現でき、またテスト用に抽象構文木のレンダラーを書くことなくテストが実行できることから、HTML を採用しています。

このドキュメントは`spec.txt`という Markdown ファイルから生成されています。変換スクリプトにはサイドバイサイドテスト用の拡張が含まれています。`tools/makespec.py`は`spec.txt`を HTML や (他の形式に変換可能な)CommonMark に変換できます。

本ドキュメントのサンプルでは、タブ文字を表現するのに → を用います。

第 2 章

まえがき

2.1 文字と行

任意の [文字列] の並びは有効な CommonMark ドキュメントです。

文字はユニコードのコードポイントです。一部のコードポイント (たとえばアクセントの組み合わせなど) は、直感的な意味での文字には対応していませんが、本仕様ではすべてのコードポイントは文字として扱われます。

本仕様では文字エンコーディングについては指定しません。行はバイトではなく [文字] で構成されているとみなされます。適合するパーサーは対応する文字エンコーディングを限定してもよい。

行はニューライン (`U+000A`) もしくはキャリッジリターン (`U+000D`) 以外の [文字] の 0 個以上のシーケンスで、その後に [行末] もしくはファイルの末尾が続くものを指します。

行末はニューライン (`U+000A`)、ニューラインの直後以外のキャリッジリターン (`U+000D`)、キャリッジリターンに続くニューラインを指します。

文字列を含まない行や空白文字 (`U+0020`) やタブ文字 (`U+0009`) のみを含む行を空行と呼びます。

本仕様では以下の文字クラスを利用します。

空白文字はスペース (`U+0020`)、タブ (`U+0009`)、ニューライン (`U+000A`)、垂直タブ (`U+000B`)、ラインフィード (`U+000C`)、キャリッジリターン (`U+000D`) を指します。

空白は 1 文字以上の [空白文字] のシーケンスです。

ユニコード空白文字は Unicode の一般カテゴリで `Zs` に分類されているコードポイントとタブ (`U+0009`)、キャリッジリターン (`U+000D`)、ニューライン (`U+000A`)、ラインフィード (`U+000C`) を指します。

ユニコード空白は 1 文字以上の [ユニコード空白文字] のシーケンスです。

スペースは `U+0020` を指します。

非空白文字は [空白文字] 以外の任意の文字を指します。

ASCII 記号文字 は !, ", #, $, %, &, ', (,), *, +, ,, -, ., / (U+0021–2F), :, ;, <, =, >, ?, @ (U+003A–0040), [, \,], ^, _, ` (U+005B–0060), {, |, }, or ~ (U+007B–007E) を指します。

記号文字は [ASCII 記号文字] もしくは Unicode の一般カテゴリで Pc, Pd, Pe, Pf, Pi, Po, or Ps に分類されているコードポイントを指します。

2.2 タブ

行内のタブは [スペース] には展開されません。ただし、空白がブロック構造の定義に役立つ場合は、タブは 4 文字のタブストップで展開されたスペースとして扱われます。

したがって、インデントコードブロックで 4 つのスペースの代わりにタブを利用できます。(ただし、行内のタブはスペースに展開されないことに注意してください。)

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

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

```
    a→a
    ^^e1^^bd^^90→a
.
<pre><code>a→a
^^e1^^bd^^90→a
</code></pre>
```

次の例では、箇条書き内の継続段落がタブでインデントされています。これは 4 つのスペースでインデントされた場合と同じ効果があります。

```
  - foo

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

```
- foo

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

通常、ブロック引用を開始する>の後にはオプションでスペースを続けることができますが、その場合はこのスペースはコンテンツの一部とはみなされません。以下のように>の後にタブが続いた場合、このタブは3つのスペースとして扱われます。これらのスペースの1つは区切り文字区切り文字の一部とみなされるため、文字列`foo`はブロック引用内で6つのスペースでインデントされているものと解釈され、2つのスペースで始まるインデントコードブロックに変換されます。

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

```
-→→foo
.
<ul>
<li>
<pre><code>  foo
</code></pre>
</li>
</ul>
```

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

```
 - foo
   - bar
→ - baz
.
<ul>
```

(次のページに続く)

(前のページからの続き)

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

```
#→Foo
.
<h1>Foo</h1>
```

```
*→*→*→
.
<hr />
```

2.3 安全でない文字

セキュリティ上の理由により、ユニコード文字 `U+0000` は REPLACEMENT CHARACTER (`U+FFFD`) に変換されます。

第 3 章

ブロックとインライン

ドキュメントはブロックのシーケンスと考えることができます。ここでいうブロックとは段落、ブロック引用、リスト、見出し、罫線、コードブロックなどの構造要素を指します。ブロック引用やリストなどのブロックは他のブロック要素を含めることができます。また、見出しや段落などのブロックはインラインコンテンツを含めることができます。インラインにはテキスト、リンク、強調されたテキスト、画像、コードスパンなどがあります。

3.1 優先順位

ブロック構造のインジケータは常にインライン構造のインジケータよりも優先されます。そのため、次の例はコードスパンを含む 1 項目の箇条書きではなく、2 項目の箇条書きと扱われます。

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

これは解析が 2 つのステップで進められることを意味します。最初のステップではドキュメントのブロック構造を識別し、次のステップでは段落や見出しなどのブロック要素内のテキストからインライン要素を解析します。2 番目のステップでは、最初のステップの最後に得られるリンク参照定義の情報を必要とします。なお、最初のステップは各行を順番に処理していく必要がありますが、各ブロック要素のインライン解析はお互いに影響を与えないため、2 番目のステップは並列化できます。

3.2 コンテナブロックとリーフブロック

ブロックは他のブロックを含むことのできるコンテナブロックと、ブロックを含むことのできないリーフブロックのふたつに分類できます。

第 4 章

リーフブロック

本章では Markdown ドキュメントを構成するさまざまな種類のリーフブロックについて説明します。

4.1 主題分割

0-3 個のスペースでインデントされた 3 つ以上の -, _ もしくは * のシーケンスは主題分割 (thematic break) を形成します。主題分割の後には任意の数のスペースやタブを続けても構いません。

```
***
---
___
.
<hr />
<hr />
<hr />
```

不適切な文字

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

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

文字数が足りない場合

```
--
**
__
.
```

(次のページに続く)

(前のページからの続き)

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

3 スペースまでのインデントは許可されます。

```
 ***
  ***
   ***
.
<hr />
<hr />
<hr />
```

しかし、4 スペースは許可されません。

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

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

3 つ以上の記号を書いてもよいです。

```
_____________________________________
.
<hr />
```

記号の間にスペースを書くこともできます。

```
 - - -
.
<hr />
```

```
 **  * ** * ** * **
.
<hr />
```

```
-     -      -      -
.
```

(次のページに続く)

(前のページからの続き)

```
<hr />
```

行末にスペースを書くこともできます。

```
- - - -
.
<hr />
```

しかし、それ以外の文字は使用できません。

```
_ _ _ _ a

a------

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

行内の [非空白文字] はすべて同じ文字である必要があります。そのため、以下は主題分割とはみなされません。

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

主題分割の前後に空白は必要ありません。

```
- foo
***
- bar
.
<ul>
<li>foo</li>
</ul>
<hr />
<ul>
<li>bar</li>
</ul>
```

主題分割は段落を中断することができます。

```
Foo
***
bar
.
```

(次のページに続く)

(前のページからの続き)

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

主題分割となる条件を満たすダッシュの行が [setext 見出し] の下線としても解釈できる場合、[setext 見出し] としての解釈が優先されます。したがって、以下の例は setext 見出しであり、段落と主題分割とはみなされません。

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

ある行が主題分割とも箇条書きとも解釈できる場合、主題分割の解釈が優先されます。

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

箇条書きの中に主題分割を配置したい場合、箇条書きとは異なる要素を使用してください。

```
- Foo
- * * *
.
<ul>
<li>Foo</li>
<li>
<hr />
</li>
</ul>
```

4.2 ATX 見出し

ATX 見出しは 1-6 個のエスケープされていない#文字からなる開始シーケンス、インラインコンテンツとして解析可能な文字列、任意の長さのエスケープされていない#文字から構成されるオプションの開始シーケンスから構成されます。開始シーケンスの後には [スペース] もしくは行末が続かなくてはなりません。終了シーケンスの前には [スペース] を挟む必要があり、終了シーケンスの後にはスペースを続けることができます。また、開始シーケンスは 0-3 スペースでインデント可能です。見出しコンテンツをインラインコンテンツとして解析する前に、先頭と末尾のスペースは削除されます。見出しレベルは開始シーケンスの#の数と同じです。

シンプルな見出しの例です。

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

7 つ以上の#記号は見出しになりません。

```
####### foo
.
<p>####### foo</p>
```

見出しが空でない場合、#文字と見出しのコンテンツの間にはすくなくとも 1 つのスペースが必要です。多くの実装ではスペースが必須ではありません。しかし、オリジナルの ATX 実装ではスペースは必須とされており、この仕様は次のような文字列を見出しとみなさないため有用です。

```
#5 bolt

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

先頭の # がエスケープされているため、見出しにはなりません:

```
\## foo
.
```

(次のページに続く)

(前のページからの続き)

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

見出しのコンテンツはインラインとして解釈されます。

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

先頭と末尾の [空白] はインライン要素の解釈時に無視されます。

```
#                  foo
.
<h1>foo</h1>
```

1-3 文字のインデントは許容されます。

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

4 つ以上のスペースでインデントした場合は見出しとみなされません。

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

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

終了シーケンスとして末尾に#文字をつけても構いません。

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

先頭と末尾の#の数は一致しなくても構いません。

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

終了シーケンスの後にスペースを入れても構いません。

```
### foo ###
.
<h3>foo</h3>
```

#の後に [空白] 以外の文字がある場合、#は終了シーケンスではなく見出しコンテンツの一部とみなされます。

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

終了シーケンスの前にはスペースが必要です。

```
# foo#
.
<h1>foo#</h1>
```

バックスラッシュでエスケープされた#文字は終了シーケンスの一部とみなされません。

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

ATX 見出しは周囲のコンテンツと空行で区切られている必要はありません。また、ATX 見出しは段落を中断することができます。

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

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

ATX見出しのコンテンツは空でも構いません。

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

4.3 Setext見出し

setext見出しは1つ以上の[非空白文字]を含み、なおかつ4スペース以上のインデントを持たない1行以上のテキストと、それに続く[setext見出し下線]から構成されます。テキスト部分はsetext見出し下線がない場合に、段落として解釈されるようなものでなければなりません。つまり、[コードフェンス]、[ATX見出し]、[引用ブロック]、[主題分割]、[箇条書き]、[HTMLブロック]と解釈できません。

setext見出し下線は=のシーケンスもしくは–のシーケンスです。3スペースまでのインデントと末尾のスペースを許容します。もし–が1つだけの場合、[setext見出し下線]ではなく空の箇条書きとして解釈されます。

見出しのレベルは[setext見出し下線]に利用されている文字の種類で決まります。=が使用されている場合はレベル1、–が使用されている場合はレベル2の見出しとしてそれぞれ扱われます。また、見出し文にはテキスト部をインラインコンテンツとして解釈したものが適用されます。

一般にsetext見出しの前後に空行を置く必要はありません。しかし、setext見出しは段落を中断することができないため、段落の後にsetext見出しを置く場合には空行を挟む必要があります。

シンプルな見出しの例です。

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

Foo *bar*
---------
.
<h1>Foo <em>bar</em></h1>
<h2>Foo <em>bar</em></h2>
```

見出し文は複数行にわたる場合があります。

```
Foo *bar
baz*
====
.
<h1>Foo <em>bar
baz</em></h1>
```

見出し文は見出しの生テキストをインラインとして解釈したものです。見出しの生テキストは各行を連結し、先頭と末尾の [空白] を除去したものです。

```
  Foo *bar
baz*→
====
.
<h1>Foo <em>bar
baz</em></h1>
```

setext 見出し下線は任意の長さをとります。

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

Foo
=
.
<h2>Foo</h2>
<h1>Foo</h1>
```

見出し文は 3 つまでインデントできます。また、下線に合わせる必要はありません。

```
   Foo
---

  Foo
-----

  Foo
  ===
.
<h2>Foo</h2>
<h2>Foo</h2>
<h1>Foo</h1>
```

4 つ以上のスペースでインデントした場合は見出しとみなされません。

```
    Foo
    ---

    Foo
---
.
<pre><code>Foo
---

Foo
</code></pre>
<hr />
```

setext 見出し下線は 3 つまでインデントできます。また、末尾のスペースを許容します。

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

4 つ以上のスペースでインデントした場合は見出しとみなされません。

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

setext 見出し下線の途中にスペースを含めることはできません。

```
Foo
= =

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

見出し文の末尾にスペースを入れても改行されません。

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

行末にバックスラッシュをおいても改行されません。

```
Foo\
----
.
<h2>Foo\</h2>
```

ブロック構造のインジケータはインライン構造のインジケータより優先されるため、以下の例は setext 見出しとして解釈されます。

```
`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>
```

setext 見出し下線は箇条書きやブロック引用の [怠惰継続行] になることはできません。

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

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

```
- Foo
---
.
<ul>
```

(次のページに続く)

(前のページからの続き)

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

段落と setext 見出しの間には空白行が必要です。さもなければ段落は見出し文の一部として扱われます。

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

しかし、一般的には setext 見出しの前後に空行は必要ありません。

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

空の setext 見出しを作ることはできません。

```

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

setext 見出しのテキスト部は段落以外のブロック構造として解釈可能であってはなりません。そのため、以下の例では setext 見出し下線ではなく主題分割として解釈されます。

```
---
---
.
<hr />
<hr />
```

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

(次のページに続く)

(前のページからの続き)

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

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

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

見出し文を`> foo`としたい場合は、バックスラッシュを使ってエスケープしてください。

```
\> foo
------
.
<h2>&gt; foo</h2>
```

互換性に関する注意: 多くの Markdown 処理系は setext 見出しが複数行にまたがることができません。しかし、解釈方法について合意はありません。

```
Foo
bar
---
baz
```

4 種類の解釈があります。

1. 段落 "Foo" と見出し "bar"、段落 "baz"
2. 段落 "Foo bar"、主題分割、段落 "baz"
3. 段落 "Foo bar --- baz"
4. 見出し "Foo bar"、段落 "baz"

我々は解釈 4 がもっとも自然で、複数行の見出しによって CommonMark の表現力が高まると考えました。著者は最初の段落の後に空行を入れることで、解釈 1 を表現できます。

```
Foo

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

解釈 2 を表現したい場合は、主題分割の前後に空行を入れます。

```
Foo
bar

---

baz
.
<p>Foo
bar</p>
<hr />
<p>baz</p>
```

もしくは [setext 見出し下線] として扱われない主題分割の記法を用います。

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

解釈 3 を表現したい場合は、バックスラッシュによるエスケープが可能です。

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

4.4 インデントコードブロック

インデントコードブロックは空行によって分割された 1 行以上の [インデントチャンク] から構成されます。インデントチャンクは非空白行のシーケンスで、各行は 4 文字以上のスペースでインデントしたものです。コードブロックの本文は各行の [行末] を含んだ内容から 4 スペース分のインデントを差し引いたものです。インデントコードブロックは [情報文字列] を持ちません。

インデントコードブロックは段落を中断できないため、段落の直後に配置する場合は空行を挟む必要があります。(コードブロックに段落が続く場合は空行は必要ありません。)

```
    a simple
      indented code block
.
<pre><code>a simple
  indented code block
</code></pre>
```

インデントがコードブロックとも [箇条書き] のコンテンツとも解釈できる場合、箇条書きの解釈が優先されます。

```
  - foo

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

```
1.  foo

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

コードブロックの本文は Markdown として解釈されません。

```
    <a/>
    *hi*
```

(次のページに続く)

(前のページからの続き)

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

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

空行によって分割された3つのチャンクの例です。

```
    chunk1

    chunk2



    chunk3
.
<pre><code>chunk1

chunk2



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

4文字を超えるスペースはたとえ空行内であってもコードブロックの本文とみなされます。

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

インデントコードブロックは段落を中断できません。(これによりインデントをぶら下げることができます。)

```
Foo
    bar

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

ただし、先頭のスペースが 4 つ未満の空白以外の行はコードブロックをすぐに終了させます。そのため、コードブロックの直後に段落が発生する可能性があります。

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

インデントコードブロックは他のブロック構造の前もしくは後に登場する可能性があります。

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

コードブロックの先頭行を 4 つ以上インデントすることもできます。

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

インデントコードブロックの先頭もしくは末尾の空白行は本文には含まれません。

```


    foo


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

行末のスペースはコードブロックの本文に含まれます。

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

4.5 フェンスドコードブロック

コードフェンスは3つ以上の連続したバックチック(`)もしくはチルダ(~)のシーケンスです。(チルダとバックチックを混在させることはできません。)フェンスドコードブロックは3つ以下のスペースでインデントされたコードフェンスで始まります。

開始コードフェンスの行にはオプションで情報文字列を書くことができます。情報文字列の前後の空白は削除されます。開始コードフェンスがバックチックの場合、[情報文字列]にはバックチックを含んでいない可能性があります。(この制限の理由は、さもなければ、インラインコードがフェンスドコードブロックの開始として誤って解釈されるためです。)

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.)

フェンスドコードブロックは段落を中断する可能性があり、前後に空白を必要としません。

フェンスドコードブロックの本文はインラインとして解釈されません。[情報文字列]の最初の単語は通常、コードサンプルの言語を指定するために使用され、`code`タグの`class`属性としてレンダリングされます。しかし、本仕様では[情報文字列]の特定の扱いは指定されません。

バックチックを使ったシンプルな例です。

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

チルダの例です。

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

バックチックが不足している例です。

~~~~
``
foo
``
.
<p><code>foo</code></p>
~~~~

終了コードフェンスは開始コードフェンスと同じ文字を使わなくてはなりません。

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

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

終了コードフェンスは少なくとも開始コードフェンスと同じ長さでなくてはなりません。

~~~~
````
aaa
```
``````
.
<pre><code>aaa
~~~~

(次のページに続く)

(前のページからの続き)

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

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

閉じられていないコードブロックはドキュメントの末尾で閉じられます。(もしくは囲んでいる [ブロック引用] または [箇条書き] の終了で閉じられます。)

~~~
```
.
<pre><code></code></pre>
~~~

~~~
`````

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

``````
> ```
> aaa

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

コードブロックはすべて空行でも構いません。

~~~
```


```
~~~

(次のページに続く)

(前のページからの続き)

````
.
<pre><code>

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

コードブロックは空でも構いません。

~~~
```
```
.
<pre><code></code></pre>
~~~

フェンスはインデントできます。開始フェンスがインデントされており、本文から同数のインデントが削除されます。

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

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

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

4 つのスペースでインデントされている場合はインデントコードブロックと解釈されます。

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

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

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

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

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

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

~~~
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:

```
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.

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

(次のページに続く)

(前のページからの続き)

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

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

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

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

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

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

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

Closing code fences cannot have [info strings]:

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

4.6 HTML blocks

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

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

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

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

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

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

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

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

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

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

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

```
<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:

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

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

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

(次のページに続く)

(前のページからの続き)

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

A block can also start with a closing tag:

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

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

```
<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:

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

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

An open tag need not be closed:

```
<div>
*foo*

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

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

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

```
<div class
foo
.
<div class
foo
```

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

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

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

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

```
<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:

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

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

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

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

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

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

```
</ins>
*bar*
.
</ins>
*bar*
```

These rules are designed to allow us to work with tags that can function as either block-level or inline-level tags. The
```

`<del>` tag is a nice example. We can surround content with `<del>` tags in three different ways. In this case, we get a raw HTML block, because the `<del>` tag is on a line by itself:

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

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

```
<del>

*foo*

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

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

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

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

A pre tag (type 1):

```
<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
```

(次のページに続く)

(前のページからの続き)

```

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

A script tag (type 1):

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

document.getElementById("demo").innerHTML = "Hello JavaScript!";
</script>
okay
.
<script type="text/javascript">
// JavaScript example

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

A style tag (type 1):

```
<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][block quotes] or [list item][list items]):

```
<style
  type="text/css">

foo
```

(次のページに続く)

(前のページからの続き)

```
.
<style
  type="text/css">

foo
```

```
> <div>
> foo

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

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

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

A comment (type 2):

```
<!-- Foo

bar
   baz -->
okay
.
<!-- Foo

bar
   baz -->
<p>okay</p>
```

A processing instruction (type 3):

```
<?php

  echo '>';

?>
okay
.
<?php

  echo '>';

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

A declaration (type 4):

```
<!DOCTYPE html>
.
<!DOCTYPE html>
```

CDATA (type 5):

```
<![CDATA[
function matchwo(a,b)
{
```

(次のページに続く)

(前のページからの続き)

```
  if (a < b && a < 0) then {
    return 1;

  } else {

    return 0;
  }
}
]]>
okay
.
<![CDATA[
function matchwo(a,b)
{
  if (a < b && a < 0) then {
    return 1;

  } else {

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

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

```
  <!-- foo -->

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

```
  <div>

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

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

```
Foo
<div>
```

(次のページに続く)

(前のページからの続き)

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

However, a following blank line is needed, except at the end of a document, and except for blocks of types 1--5, [above][HTML block]:

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

HTML blocks of type 7 cannot interrupt a paragraph:

```
Foo
<a href="bar">
baz
.
<p>Foo
<a href="bar">
baz</p>
```

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

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

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

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

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

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

Compare:

```
<div>

*Emphasized* text.

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

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

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

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

```
<table>

<tr>

<td>
Hi
</td>

</tr>

</table>
.
<table>
<tr>
```

(次のページに続く)

(前のページからの続き)

```
<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:

```
<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][HTML blocks], raw HTML blocks starting with `<pre>` *can* contain blank lines.

4.7 Link reference definitions

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

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

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

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

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

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

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

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

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

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

The title may extend over multiple lines:

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

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

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

```
[foo]: /url 'title

with blank line'

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

The title may be omitted:

```
[foo]:
/url

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

The link destination may not be omitted:

```
[foo]:

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

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

```
[foo]: <>

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

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

```
[foo]: <bar>(baz)

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

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

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

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

A link can come before its corresponding definition:

```
[foo]

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

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

```
[foo]

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

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

```
[FOO]: /url

[Foo]
.
<p><a href="/url">Foo</a></p>
```

```
[ΑΓΩ]: /φου

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

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

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

Here is another one:

```
[
foo
```

(次のページに続く)

(前のページからの続き)

```
]: /url
bar
.
<p>bar</p>
```

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

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

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

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

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

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

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

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

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

[foo]
.
<pre><code>[foo]: /url
</code></pre>
<p>[foo]</p>
~~~

A [link reference definition] cannot interrupt a paragraph.

```
Foo
[bar]: /baz

[bar]
```

(次のページに続く)

(前のページからの続き)

```
.
<p>Foo
[bar]: /baz</p>
<p>[bar]</p>
```

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

```
# [Foo]
[foo]: /url
> bar
.
<h1><a href="/url">Foo</a></h1>
<blockquote>
<p>bar</p>
</blockquote>
```

```
[foo]: /url
bar
===
[foo]
.
<h1>bar</h1>
<p><a href="/url">foo</a></p>
```

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

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

```
[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:

```
[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:

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

4.8 Paragraphs

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

A simple example with two paragraphs:

```
aaa

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

Paragraphs can contain multiple lines, but no blank lines:

```
aaa
bbb

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

Multiple blank lines between paragraph have no effect:

```
aaa


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

Leading spaces are skipped:

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

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

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

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

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

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

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

```
aaa
bbb
```

(次のページに続く)

(前のページからの続き)

```
.
<p>aaa<br />
bbb</p>
```

4.9 Blank lines

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

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

```


aaa


# aaa


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

第 5 章

Container blocks

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

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

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

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

5.1 Block quotes

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

The following rules define [block quotes]:

1. **Basic case.** If a string of lines *Ls* constitute a sequence of blocks *Bs*, then the result of prepending a [block quote marker] to the beginning of each line in *Ls* is a *block quote* containing *Bs*.

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

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

Nothing else counts as a *block quote*.

Here is a simple example:

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

The spaces after the > characters can be omitted:

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

The > characters can be indented 1-3 spaces:

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

Four spaces gives us a code block:

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

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

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

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

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

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

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

without changing the meaning:

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

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

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

then the block quote ends after the first line:

```
> - foo
- bar
```

(次のページに続く)

(前のページからの続き)

```
.
<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:

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

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

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

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

To see why, note that in

```
> foo
>     - bar
```

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

A block quote can be empty:

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

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

A block quote can have initial or final blank lines:

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

A blank line always separates block quotes:

```
> 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:

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

To get a block quote with two paragraphs, use:

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

Block quotes can interrupt paragraphs:

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

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

```
> 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:

```
> bar
baz
```

(次のページに続く)

(前のページからの続き)

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

```
> bar

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

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

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

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

```
>>> foo
> bar
>>baz
.
<blockquote>
<blockquote>
<blockquote>
```

(次のページに続く)

(前のページからの続き)

```
<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 >:

```
>     code

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

5.2 List items

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

A bullet list marker is a `-`, `+`, or `*` character.

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

The following rules define [list items]:

1. **Basic case.** If a sequence of lines *Ls* constitute a sequence of blocks *Bs* starting with a [non-whitespace character], and *M* is a list marker of width *W* followed by 1 âL’d’ *N* âL’d’ 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][thematic breaks] then that line is not a list item.

For example, let *Ls* be the lines

```
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*:

```
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:

```
- one

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

```
- one

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

```
 -    one

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

```
 -    one

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

It is tempting to think of this in terms of columns: the continuation blocks must be indented at least to the column of the first [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:

```
   > > 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:

```
>>- 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:

```
-one

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

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

```
- foo


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

A list item may contain any kind of block:

~~~
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.

```
- Foo

      bar


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

(次のページに続く)

(前のページからの続き)

```
<pre><code>bar


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

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

```
123456789. ok
.
<ol start="123456789">
<li>ok</li>
</ol>
```

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

A start number may begin with 0s:

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

```
003. ok
.
<ol start="3">
<li>ok</li>
</ol>
```

A start number may not be negative:

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

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

number, based on the ordered list marker.

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

```
- foo

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

And in this case it is 11 spaces:

```
  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:

```
    indented code

paragraph

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

```
1.     indented code

```

(次のページに続く)

(前のページからの続き)

```
   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:

```
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:

```
   foo

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

```
-    foo
```

(次のページに続く)

(前のページからの続き)

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

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

```
-  foo

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

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

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

~~~
-
  foo
-
  ```
  bar
  ```
-
      baz
.
<ul>
<li>foo</li>
<li>
<pre><code>bar
</code></pre>
</li>
<li>
~~~

(次のページに続く)

(前のページからの続き)

```
<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:

```
-
  foo
.
<ul>
<li>foo</li>
</ul>
```

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

```
-

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

Here is an empty bullet list item:

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

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

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

(次のページに続く)

(前のページからの続き)

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

Here is an empty ordered list item:

```
1. foo
2.
3. bar
.
<ol>
<li>foo</li>
<li></li>
<li>bar</li>
</ol>
```

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

```
*
.
<ul>
<li></li>
</ul>
```

However, an empty list item cannot interrupt a paragraph:

```
foo
*

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

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

Indented one space:

```
 1.  A paragraph
     with two lines.

```

(次のページに続く)

(前のページからの続き)

```
         indented code

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

Indented two spaces:

```
  1.  A paragraph
      with two lines.

          indented code

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

Indented three spaces:

```
   1.  A paragraph
       with two lines.

           indented code

       > A block quote.
.
<ol>
```

(次のページに続く)

(前のページからの続き)

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

Four spaces indent gives a code block:

```
    1.  A paragraph
        with two lines.

            indented code

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

        indented code

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

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

Here is an example with [lazy continuation lines]:

```
  1.  A paragraph
with two lines.

          indented code

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

(次のページに続く)

(前のページからの続き)

```
</code></pre>
<blockquote>
<p>A block quote.</p>
</blockquote>
</li>
</ol>
```

Indentation can be partially deleted:

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

These examples show how laziness can work in nested structures:

```
> 1. > Blockquote
continued here.
.
<blockquote>
<ol>
<li>
<blockquote>
<p>Blockquote
continued here.</p>
</blockquote>
</li>
</ol>
</blockquote>
```

```
> 1. > Blockquote
> continued here.
.
<blockquote>
<ol>
<li>
<blockquote>
<p>Blockquote
continued here.</p>
</blockquote>
</li>
</ol>
</blockquote>
```

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

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

So, in this case we need two spaces indent:

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

One is not enough:

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

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

```
10) foo
    - bar
.
<ol start="10">
```

(次のページに続く)

(前のページからの続き)

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

Three is not enough:

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

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

```
- - foo
.
<ul>
<li>
<ul>
<li>foo</li>
</ul>
</li>
</ul>
```

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

A list item can contain a heading:

```
- # Foo
- Bar
  ---
  baz
.
<ul>
<li>
<h1>Foo</h1>
</li>
<li>
<h2>Bar</h2>
baz</li>
</ul>
```

5.2.1 Motivation

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

1. "List markers typically start at the left margin, but may be indented by up to three spaces. List markers must be followed by one or more spaces or a tab."

2. "To make lists look nice, you can wrap items with hanging indents.... But if you don't want to, you don't have to."

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

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

5. "To put a blockquote within a list item, the blockquote's > delimiters need to be indented."

6. "To put a code block within a list item, the code block needs to be indented twice — 8 spaces or two tabs."

These rules specify that a paragraph under a list item must be indented four spaces (presumably, from the left margin, rather than the start of the list marker, but this is not said), and that code under a list item must be indented eight spaces instead of the usual four. They also say that a block quote must be indented, but not by how much; however, the example given has four spaces indentation. Although nothing is said about other kinds of block-level content, it is certainly reasonable to infer that *all* block elements under a list item, including other lists, must be indented four spaces. This principle has been called the *four-space rule*.

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

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

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

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

```
- foo

  bar

  - baz
```

should be parsed as two lists with an intervening paragraph,

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

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

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

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

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

```
   - one

  two
```

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

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

and similarly

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

as

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

This is extremely unintuitive.

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

```
 10. foo

   bar
```

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

```
1.  foo

        indented code
```

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

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

5.3 Lists

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

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

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

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

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

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

```
- foo
- bar
+ baz
.
<ul>
<li>foo</li>
<li>bar</li>
</ul>
<ul>
```

(次のページに続く)

(前のページからの続き)

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

```
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:

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

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

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

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

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

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

Second, we are attracted to a

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

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

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

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

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

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

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

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

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

We may still get an unintended result in cases like

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

but this rule should prevent most spurious list captures.

There can be any number of blank lines between items:

```
- foo

- bar


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

```
- 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:

```
- foo
- bar

```

(次のページに続く)

(前のページからの続き)

```
<!-- -->

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

```
-   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:

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

(次のページに続く)

(前のページからの続き)

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

```
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:

```
- 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.

```
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:

```
- a
- b

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

So is this, with a empty second item:

```
* a
*

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

(次のページに続く)

(前のページからの続き)

```
</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:

```
- a
- b

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

```
- a
- b

  [ref]: /url
- d
.
<ul>
<li>
<p>a</p>
</li>
<li>
<p>b</p>
</li>
<li>
<p>d</p>
</li>
</ul>
```

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

~~~
- 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:

```
- a
  - b

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

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

```
* a
  > b
  >
* c
```

(次のページに続く)

(前のページからの続き)

```
.
<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:

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

A single-paragraph list is tight:

```
- a
.
<ul>
<li>a</li>
</ul>
```

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

(次のページに続く)

(前のページからの続き)

```
</ul>
```

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

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

   bar
.
<ol>
<li>
<pre><code>foo
</code></pre>
<p>bar</p>
</li>
</ol>
~~~~

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

```
* foo
  * bar

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

```
- a
  - b
  - c

- d
  - e
  - f
.
<ul>
<li>
<p>a</p>
<ul>
```

(次のページに続く)

(前のページからの続き)

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

第 6 章

Inlines

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

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

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

6.1 Backslash escapes

Any ASCII punctuation character may be backslash-escaped:

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

Backslashes before other characters are treated as literal backslashes:

```
\→\A\a\ \3\φ\^^c2^^ab
.
<p>\→\A\a\ \3\φ\^^c2^^ab</p>
```

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

```
\*not emphasized*
\<br/> not a tag
\[not a link](/foo)
\`not code`
1\. not a list
\* not a list
```

(次のページに続く)

(前のページからの続き)

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

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

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

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

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

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

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

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

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

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

```
<a href="/bar\/)">
.
<a href="/bar\/)">
```

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

```
[foo](/bar\* "ti\*tle")
.
<p><a href="/bar*" title="ti*tle">foo</a></p>
```

```
[foo]

[foo]: /bar\* "ti\*tle"
.
<p><a href="/bar*" title="ti*tle">foo</a></p>
```

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

6.2 Entity and numeric character references

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

- Entity and character references are not recognized in code blocks and code spans.
- Entity and character references cannot stand in place of special characters that define structural elements in CommonMark. For example, although `&#42;` can be used in place of a literal `*` character, `&#42;` cannot replace `*` in emphasis delimiters, bullet list markers, or thematic breaks.

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

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

```
&nbsp; &amp; &copy; &AElig; &Dcaron;
&frac34; &HilbertSpace; &DifferentialD;
&ClockwiseContourIntegral; &ngE;
.
<p>^^c2^^a0 &amp; ^^c2^^a9 ^^c3^^86 ^^c4^^8e
^^c2^^be ^^e2^^84^^8b ^^e2^^85^^86
^^e2^^88^^b2 ≧^^cc^^b8</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`.

```
&#35; &#1234; &#992; &#0;
.
<p># ^^d3^^92 ^^cf^^a0 ^^ef^^bf^^bd</p>
```

Hexadecimal numeric character references consist of `&#` + either `X` or `x` + a string of 1-6 hexadecimal digits + `;`. They too are parsed as the corresponding Unicode character (this time specified with a hexadecimal numeral instead of decimal).

```
&#X22; &#XD06; &#xcab;
.
<p>&quot; ^^e0^^b4^^86 ^^e0^^b2^^ab</p>
```

Here are some nonentities:

```
&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:

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

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

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

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

```
<a href="&ouml;&ouml;.html">
.
<a href="&ouml;&ouml;.html">
```

```
[foo](/f&ouml;&ouml; "f&ouml;&ouml;")
.
<p><a href="/f%C3%B6%C3%B6" title="f^^c3^^b6^^c3^^b6">foo</a></p>
```

```
[foo]

[foo]: /f&ouml;&ouml; "f&ouml;&ouml;"
.
<p><a href="/f%C3%B6%C3%B6" title="f^^c3^^b6^^c3^^b6">foo</a></p>
```

~~~
``` f&ouml;&ouml;
foo
```
.
<pre><code class="language-f^^c3^^b6^^c3^^b6">foo
</code></pre>
~~~

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

```
&#42; foo

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

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

bar</p>
```

```
&#9;foo
.
<p>→foo</p>
```

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

6.3 Code spans

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

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

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

This is a simple code span:

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

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

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

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

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

Note that only *one* space is stripped:

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

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

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

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

```
`^^c2^^a0b^^c2^^a0`
.
<p><code>^^c2^^a0b^^c2^^a0</code></p>
```

No stripping occurs if the code span contains only spaces:

```
`^^c2^^a0`
`  `
.
<p><code>^^c2^^a0</code>
<code>  </code></p>
```

[Line endings] are treated like spaces:

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

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

Interior spaces are not collapsed:

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

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

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

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

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

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

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

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

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

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

And this is not parsed as a link:

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

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

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

But this is an HTML tag:

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

And this is code:

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

But this is an autolink:

```
<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:

~~~
```foo``
.
<p>```foo``</p>
~~~

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

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

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

6.4 Emphasis and strong emphasis

John Gruber's original Markdown syntax description says:

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

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

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

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

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

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

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

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

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

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

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

Here are some examples of delimiter runs.

- left-flanking but not right-flanking:

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

- right-flanking but not left-flanking:

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

- Both left and right-flanking:

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

- Neither left nor right-flanking:

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

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

The following rules define emphasis and strong emphasis:

1. A single `*` character can open emphasis iff (if and only if) it is part of a [left-flanking delimiter run].

2. A single _ character [can open emphasis] iff it is part of a [left-flanking delimiter run] and either (a) not part of a [right-flanking delimiter run] or (b) part of a [right-flanking delimiter run] preceded by punctuation.

3. A single `*` character can close emphasis iff it is part of a [right-flanking delimiter run].

4. A single _ character [can close emphasis] iff it is part of a [right-flanking delimiter run] and either (a) not part of a [left-flanking delimiter run] or (b) part of a [left-flanking delimiter run] followed by punctuation.

5. A double `**` can open strong emphasis iff it is part of a [left-flanking delimiter run].

6. A double __ [can open strong emphasis] iff it is part of a [left-flanking delimiter run] and either (a) not part of a [right-flanking delimiter run] or (b) part of a [right-flanking delimiter run] preceded by punctuation.

7. A double `**` can close strong emphasis iff it is part of a [right-flanking delimiter run].

8. A double __ [can close strong emphasis] iff it is part of a [right-flanking delimiter run] and either (a) not part of a [left-flanking delimiter run] or (b) part of a [left-flanking delimiter run] followed by punctuation.

9. Emphasis begins with a delimiter that [can open emphasis] and ends with a delimiter that [can close emphasis], and that uses the same character (_ or `*`) as the opening delimiter. The opening and closing delimiters must belong to separate [delimiter runs]. If one of the delimiters can both open and close emphasis, then the sum of the lengths of the delimiter runs containing the opening and closing delimiters must not be a multiple of 3 unless both lengths are multiples of 3.

10. Strong emphasis begins with a delimiter that [can open strong emphasis] and ends with a delimiter that [can close strong emphasis], and that uses the same character (_ or `*`) as the opening delimiter. The opening and closing delimiters must belong to separate [delimiter runs]. If one of the delimiters can both open and close strong emphasis, then the sum of the lengths of the delimiter runs containing the opening and closing delimiters must not be a multiple of 3 unless both lengths are multiples of 3.

11. A literal `*` character cannot occur at the beginning or end of `*`-delimited emphasis or `**`-delimited strong emphasis, unless it is backslash-escaped.

12. A literal _ character cannot occur at the beginning or end of _-delimited emphasis or __-delimited strong emphasis, unless it is backslash-escaped.

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

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

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

15. When two potential emphasis or strong emphasis spans overlap, so that the second begins before the first ends and ends after the first ends, the first takes precedence. Thus, for example, `*foo _bar* baz_` is parsed as `<em>foo _bar</em> baz_` rather than `*foo <em>bar* baz</em>`.

16. When there are two potential emphasis or strong emphasis spans with the same closing delimiter, the shorter one (the one that opens later) takes precedence. Thus, for example, `**foo **bar baz**` is parsed as `**foo <strong>bar baz</strong>` rather than `<strong>foo **bar baz</strong>`.

17. Inline code spans, links, images, and HTML tags group more tightly than emphasis. So, when there is a choice between an interpretation that contains one of these elements and one that does not, the former always wins. Thus, for example, `*[foo*](bar)` is parsed as `*<a href="bar">foo*</a>` rather than as

<em>[foo</em>](bar).

These rules can be illustrated through a series of examples.

Rule 1:

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

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

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

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

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

Unicode nonbreaking spaces count as whitespace, too:

```
*^^c2^^a0a^^c2^^a0*
.
<p>*^^c2^^a0a^^c2^^a0*</p>
```

Intraword emphasis with * is permitted:

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

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

Rule 2:

```
_foo bar_
.
<p><em>foo bar</em></p>
```

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

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

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

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

Emphasis with _ is not allowed inside words:

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

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

```
пристаням_стремятся_
.
<p>пристаням_стремятся_</p>
```

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

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

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

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

Rule 3:

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

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

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

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

A newline also counts as whitespace:

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

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

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

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

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

Intraword emphasis with `*` is allowed:

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

Rule 4:

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

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

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

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

This is emphasis within emphasis:

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

Intraword emphasis is disallowed for _:

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

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

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

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

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

Rule 5:

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

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

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

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

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

Intraword strong emphasis with `**` is permitted:

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

Rule 6:

```
__foo bar__
.
<p><strong>foo bar</strong></p>
```

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

```
__ foo bar__
.
<p>__ foo bar__</p>
```

A newline counts as whitespace:

```
__
foo bar__
.
<p>__
foo bar__</p>
```

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

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

Intraword strong emphasis is forbidden with __:

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

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

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

```
__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:

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

Rule 7:

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

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

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

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

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

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

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

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

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

Intraword emphasis:

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

Rule 8:

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

```
__foo bar __
.
<p>__foo bar __</p>
```

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

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

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

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

Intraword strong emphasis is forbidden with __:

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

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

```
__foo__bar__baz__
.
<p><strong>foo__bar__baz</strong></p>
```

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

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

Rule 9:

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

```
*foo [bar](/url)*
.
<p><em>foo <a href="/url">bar</a></em></p>
```

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

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

```
_foo __bar__ baz_
.
<p><em>foo <strong>bar</strong> baz</em></p>
```

```
_foo _bar_ baz_
.
<p><em>foo <em>bar</em> baz</em></p>
```

```
__foo_ bar_
.
<p><em><em>foo</em> bar</em></p>
```

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

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

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

Note that in the preceding case, the interpretation

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

is precluded by the condition that a delimiter that can both open and close (like the `*` after `foo`) cannot form emphasis if the sum of the lengths of the delimiter runs containing the opening and closing delimiters is a multiple of 3 unless both lengths are multiples of 3.

For the same reason, we don't get two consecutive emphasis sections in this example:

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

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

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

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

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

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

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

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

Indefinite levels of nesting are possible:

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

```
*foo [*bar*](/url)*
.
<p><em>foo <a href="/url"><em>bar</em></a></em></p>
```

There can be no empty emphasis or strong emphasis:

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

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

Rule 10:

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

```
**foo [bar](/url)**
.
<p><strong>foo <a href="/url">bar</a></strong></p>
```

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

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

```
__foo _bar_ baz__
.
<p><strong>foo <em>bar</em> baz</strong></p>
```

```
__foo __bar__ baz__
.
<p><strong>foo <strong>bar</strong> baz</strong></p>
```

```
____foo__ bar__
.
<p><strong><strong>foo</strong> bar</strong></p>
```

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

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

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

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

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

Indefinite levels of nesting are possible:

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

```
**foo [*bar*](/url)**
.
<p><strong>foo <a href="/url"><em>bar</em></a></strong></p>
```

There can be no empty emphasis or strong emphasis:

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

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

Rule 11:

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

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

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

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

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

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

Note that when delimiters do not match evenly, Rule 11 determines that the excess literal * characters will appear outside of the emphasis, rather than inside it:

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

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

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

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

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

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

Rule 12:

```
foo ___
.
<p>foo ___</p>
```

```
foo _\__
.
<p>foo <em>_</em></p>
```

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

```
foo _____
.
<p>foo _____</p>
```

```
foo __\___
.
<p>foo <strong>_</strong></p>
```

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

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

Note that when delimiters do not match evenly, Rule 12 determines that the excess literal _ characters will appear outside of the emphasis, rather than inside it:

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

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

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

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

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

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

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

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

```
__foo__
.
<p><strong>foo</strong></p>
```

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

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

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

```
____foo____
.
<p><strong><strong>foo</strong></strong></p>
```

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

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

Rule 14:

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

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

Rule 15:

```
*foo _bar* baz_
.
<p><em>foo _bar</em> baz_</p>
```

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

Rule 16:

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

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

Rule 17:

```
*[bar*](/url)
.
<p>*<a href="/url">bar*</a></p>
```

```
_foo [bar_](/url)
.
<p>_foo <a href="/url">bar_</a></p>
```

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

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

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

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

```
_a `_`_
.
<p><em>a <code>_</code></em></p>
```

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

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

6.5 Links

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

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

- Links may not contain other links, at any level of nesting. If multiple otherwise valid link definitions appear nested inside each other, the inner-most definition is used.
- Brackets are allowed in the [link text] only if (a) they are backslash-escaped or (b) they appear as a matched pair of brackets, with an open bracket `[`, a sequence of zero or more inlines, and a close bracket `]`.
- Backtick [code spans], [autolinks], and raw [HTML tags] bind more tightly than the brackets in link text. Thus, for example, ``[foo`]` `` could not be a link text, since the second `]` is part of a code span.
- The brackets in link text bind more tightly than markers for [emphasis and strong emphasis]. Thus, for example, `*[foo*](url)` is a link.

A link destination consists of either

- a sequence of zero or more characters between an opening < and a closing > that contains no line breaks or unescaped < or > characters, or
- a nonempty sequence of characters that does not start with <, does not include ASCII space or control characters, and includes parentheses only if (a) they are backslash-escaped or (b) they are part of a balanced pair of unescaped parentheses. (Implementations may impose limits on parentheses nesting to avoid performance issues, but at least three levels of nesting should be supported.)

A link title consists of either

- a sequence of zero or more characters between straight double-quote characters (`"`), including a `"` character only if it is backslash-escaped, or
- a sequence of zero or more characters between straight single-quote characters (`'`), including a `'` character only if it is backslash-escaped, or
- a sequence of zero or more characters between matching parentheses (`(...)`), including a `(` or `)` character only if it is backslash-escaped.

Although [link titles] may span multiple lines, they may not contain a [blank line].

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

Here is a simple inline link:

```
[link](/uri "title")
.
<p><a href="/uri" title="title">link</a></p>
```

The title may be omitted:

```
[link](/uri)
.
<p><a href="/uri">link</a></p>
```

Both the title and the destination may be omitted:

```
[link]()
.
<p><a href="">link</a></p>
```

```
[link](<>)
.
<p><a href="">link</a></p>
```

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

```
[link](/my uri)
.
<p>[link](/my uri)</p>
```

```
[link](</my uri>)
.
<p><a href="/my%20uri">link</a></p>
```

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

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

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

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

```
[a](<b)c>)
.
<p><a href="b)c">a</a></p>
```

Pointy brackets that enclose links must be unescaped:

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

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

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

(次のページに続く)

(前のページからの続き)

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

Parentheses inside the link destination may be escaped:

```
[link](\(foo\))
.
<p><a href="(foo)">link</a></p>
```

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

```
[link](foo(and(bar)))
.
<p><a href="foo(and(bar))">link</a></p>
```

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

```
[link](foo\(and\(bar\))
.
<p><a href="foo(and(bar)">link</a></p>
```

```
[link](<foo(and(bar)>)
.
<p><a href="foo(and(bar)">link</a></p>
```

Parentheses and other symbols can also be escaped, as usual in Markdown:

```
[link](foo\)\:)
.
<p><a href="foo):">link</a></p>
```

A link can contain fragment identifiers and queries:

```
[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>
```

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

```
[link](foo\bar)
.
<p><a href="foo%5Cbar">link</a></p>
```

URL-escaping should be left alone inside the destination, as all URL-escaped characters are also valid URL characters. Entity and numerical character references in the destination will be parsed into the corresponding Unicode code points, as usual. These may be optionally URL-escaped when written as HTML, but this spec does not enforce any particular policy for rendering URLs in HTML or other formats. Renderers may make different decisions about how to escape or normalize URLs in the output.

```
[link](foo%20b&auml;)
.
<p><a href="foo%20b%C3%A4">link</a></p>
```

Note that, because titles can often be parsed as destinations, if you try to omit the destination and keep the title, you'll get unexpected results:

```
[link]("title")
.
<p><a href="%22title%22">link</a></p>
```

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

```
[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:

```
[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.

```
[link](/url^^c2^^a0"title")
.
<p><a href="/url%C2%A0%22title%22">link</a></p>
```

Nested balanced quotes are not allowed without escaping:

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

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

```
[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.)

[Whitespace] is allowed around the destination and title:

```
[link](   /uri
  "title"  )
.
<p><a href="/uri" title="title">link</a></p>
```

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

```
[link] (/uri)
.
<p>[link] (/uri)</p>
```

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

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

```
[link] bar](/uri)
.
<p>[link] bar](/uri)</p>
```

```
[link [bar](/uri)
.
<p>[link <a href="/uri">bar</a></p>
```

```
[link \[bar](/uri)
.
<p><a href="/uri">link [bar</a></p>
```

The link text may contain inline content:

```
[link *foo **bar** `#`*](/uri)
.
<p><a href="/uri">link <em>foo <strong>bar</strong> <code>#</code></em></a></p>
```

```
[![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.

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

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

```
![[[foo](uri1)](uri2)](uri3)
.
<p><img src="uri3" alt="[foo](uri2)" /></p>
```

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

```
*[foo*](/uri)
.
<p>*<a href="/uri">foo*</a></p>
```

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

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

```
*foo [bar* baz]
.
<p><em>foo [bar</em> baz]</p>
```

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

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

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

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

There are three kinds of reference links: *full*, *collapsed*, and *shortcut*.

A full reference link consists of a [link text] immediately followed by a [link label] that [matches] a [link reference definition] elsewhere in the document.

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

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

The contents of the first link label are parsed as inlines, which are used as the link's text. The link's URI and title are provided by the matching [link reference definition].

Here is a simple example:

```
[foo][bar]

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

The rules for the [link text] are the same as with [inline links]. Thus:

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

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

[ref]: /uri
```

(次のページに続く)

(前のページからの続き)

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

```
[link \[bar][ref]

[ref]: /uri
.
<p><a href="/uri">link [bar</a></p>
```

The link text may contain inline content:

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

[ref]: /uri
.
<p><a href="/uri">link <em>foo <strong>bar</strong> <code>#</code></em></a></p>
```

```
[![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.

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

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

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

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

(In the examples above, we have two [shortcut reference links] instead of one [full reference link].)

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

```
*[foo*][ref]

[ref]: /uri
.
<p>*<a href="/uri">foo*</a></p>
```

```
[foo *bar][ref]

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

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

```
[foo <bar attr="][ref]">

[ref]: /uri
.
<p>[foo <bar attr="][ref]"></p>
```

```
[foo`][ref]`

[ref]: /uri
.
<p>[foo<code>][ref]</code></p>
```

```
[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>
```

Matching is case-insensitive:

```
[foo][BaR]

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

Unicode case fold is used:

```
[Толпой][Толпой] is a Russian word.

[ТОЛПОЙ]: /url
.
<p><a href="/url">Толпой</a> is a Russian word.</p>
```

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

```
[Foo
  bar]: /url

[Baz][Foo bar]
.
<p><a href="/url">Baz</a></p>
```

No [whitespace] is allowed between the [link text] and the [link label]:

```
[foo] [bar]

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

```
[foo]
[bar]

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

This is a departure from John Gruber's original Markdown syntax description, which explicitly allows whitespace between the link text and the link label. It brings reference links in line with [inline links], which (according to both original Markdown and this spec) cannot have whitespace after the link text. More importantly, it prevents inadvertent capture of consecutive [shortcut reference links]. If whitespace is allowed between the link text and the link label, then in the following we will have a single reference link, not two shortcut reference links, as intended:

```
[foo]
[bar]

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

(Note that [shortcut reference links] were introduced by Gruber himself in a beta version of `Markdown.pl`, but never included in the official syntax description. Without shortcut reference links, it is harmless to allow space between the link text and link label; but once shortcut references are introduced, it is too dangerous to allow this, as it frequently leads to unintended results.)

When there are multiple matching [link reference definitions], the first is used:

```
[foo]: /url1

[foo]: /url2

```

(次のページに続く)

(前のページからの続き)

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

Note that matching is performed on normalized strings, not parsed inline content. So the following does not match, even though the labels define equivalent inline content:

```
[bar][foo\!]

[foo!]: /url
.
<p>[bar][foo!]</p>
```

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

```
[foo][ref[]

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

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

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

```
[[[foo]]]

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

```
[foo][ref\[]

[ref\[]: /uri
.
<p><a href="/uri">foo</a></p>
```

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

```
[bar\\]: /uri

[bar\\]
.
<p><a href="/uri">bar\</a></p>
```

A [link label] must contain at least one [non-whitespace character]:

```
[]

[]: /uri
.
<p>[]</p>
<p>[]: /uri</p>
```

```
[
 ]

[
 ]: /uri
.
<p>[
]</p>
<p>[
]: /uri</p>
```

A collapsed reference link consists of a [link label] that [matches] a [link reference definition] elsewhere in the document, followed by the string `[]`. The contents of the 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]`.

```
[foo][]

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

```
[*foo* bar][]

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

The link labels are case-insensitive:

```
[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:

```
[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][]`.

```
[foo]

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

```
[*foo* bar]

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

```
[[*foo* bar]]

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

```
[[bar [foo]

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

The link labels are case-insensitive:

```
[Foo]

[foo]: /url "title"
.
<p><a href="/url" title="title">Foo</a></p>
```

A space after the link text should be preserved:

```
[foo] bar

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

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

```
\[foo]

[foo]: /url "title"
.
<p>[foo]</p>
```

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

```
[foo*]: /url

*[foo*]
.
<p>*<a href="/url">foo*</a></p>
```

Full and compact references take precedence over shortcut references:

```
[foo][bar]

[foo]: /url1
[bar]: /url2
.
<p><a href="/url2">foo</a></p>
```

```
[foo][]

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

Inline links also take precedence:

```
[foo]()

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

```
[foo](not a link)

[foo]: /url1
.
<p><a href="/url1">foo</a>(not a link)</p>
```

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

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

[baz]: /url
.
<p>[foo]<a href="/url">bar</a></p>
```

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

```
[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):

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

[baz]: /url1
[foo]: /url2
.
<p>[foo]<a href="/url1">bar</a></p>
```

6.6 Images

Syntax for images is like the syntax for links, with one difference. Instead of [link text], we have an image description. The rules for this are the same as for [link text], except that (a) an image description starts with `![` rather than `[`, and (b) an image description may contain links. An image description has inline elements as its contents. When an image is rendered to HTML, this is standardly used as the image's `alt` attribute.

```
![foo](/url "title")
.
<p><img src="/url" alt="foo" title="title" /></p>
```

```
![foo *bar*]

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

```
![foo ![bar](/url)](/url2)
.
<p><img src="/url2" alt="foo bar" /></p>
```

```
![foo [bar](/url)](/url2)
.
<p><img src="/url2" alt="foo bar" /></p>
```

Though this spec is concerned with parsing, not rendering, it is recommended that in rendering to HTML, only the plain string content of the [image description] be used. Note that in the above example, the alt attribute's value is `foo bar`, not `foo [bar](/url)` or `foo <a href="/url">bar</a>`. Only the plain string content is rendered, without formatting.

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

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

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

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

```
![foo](train.jpg)
.
<p><img src="train.jpg" alt="foo" /></p>
```

```
My ![foo bar](/path/to/train.jpg  "title"   )
.
<p>My <img src="/path/to/train.jpg" alt="foo bar" title="title" /></p>
```

```
![foo](<url>)
.
<p><img src="url" alt="foo" /></p>
```

```
![](/url)
.
<p><img src="/url" alt="" /></p>
```

Reference-style:

```
![foo][bar]

[bar]: /url
.
<p><img src="/url" alt="foo" /></p>
```

```
![foo][bar]

[BAR]: /url
.
<p><img src="/url" alt="foo" /></p>
```

Collapsed:

```
![foo][]

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

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

[*foo* bar]: /url "title"
.
<p><img src="/url" alt="foo bar" title="title" /></p>
```

The labels are case-insensitive:

```
![Foo][]

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

(次のページに続く)

(前のページからの続き)

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

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

```
![foo]
[]

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

Shortcut:

```
![foo]

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

```
![*foo* bar]

[*foo* bar]: /url "title"
.
<p><img src="/url" alt="foo bar" title="title" /></p>
```

Note that link labels cannot contain unescaped brackets:

```
![[foo]]

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

The link labels are case-insensitive:

```
![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 `[`:

```
!\[foo]

[foo]: /url "title"
.
<p>![foo]</p>
```

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

```
\![foo]

[foo]: /url "title"
.
<p>!<a href="/url" title="title">foo</a></p>
```

6.7 Autolinks

Autolinks are absolute URIs and email addresses inside < and >. They are parsed as links, with the URL or email address as the link label.

A URI autolink consists of <, followed by an [absolute URI] followed by >. It is parsed as a link to the URI, with the URI as the link's label.

An absolute URI, for these purposes, consists of a [scheme] followed by a colon (`:`) followed by zero or more characters other than ASCII [whitespace] and control characters, <, and >. If the URI includes these characters, they must be percent-encoded (e.g. `%20` for a space).

For purposes of this spec, a scheme is any sequence of 2--32 characters beginning with an ASCII letter and followed by any combination of ASCII letters, digits, or the symbols plus ("+"), period ("."), or hyphen ("-").

Here are some valid autolinks:

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

```
<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>
```

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

Uppercase is also fine:

```
<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:

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

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

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

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

Spaces are not allowed in autolinks:

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

Backslash-escapes do not work inside autolinks:

```
<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.

An email address, for these purposes, is anything that matches the non-normative regex from the HTML5 spec:

```
/^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?
(?:\.[a-zA-Z0-9](?:[a-zA-Z0-9-]{0,61}[a-zA-Z0-9])?)*$/
```

Examples of email autolinks:

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

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

Backslash-escapes do not work inside email autolinks:

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

These are not autolinks:

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

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

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

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

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

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

6.8 Raw HTML

Text between < and > that looks like an HTML tag is parsed as a raw HTML tag and will be rendered in HTML without escaping. Tag and attribute names are not limited to current HTML tags, so custom tags (and even, say, DocBook tags) may be used.

Here is the grammar for tags:

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

An attribute consists of [whitespace], an [attribute name], and an optional [attribute value specification].

An attribute name consists of an ASCII letter, _, or `:`, followed by zero or more ASCII letters, digits, _, `.`, `:`, or `-`. (Note: This is the XML specification restricted to ASCII. HTML5 is laxer.)

An attribute value specification consists of optional [whitespace], a = character, optional [whitespace], and an [attribute value].

An attribute value consists of an [unquoted attribute value], a [single-quoted attribute value], or a [double-quoted attribute value].

An unquoted attribute value is a nonempty string of characters not including [whitespace], `"`, `'`, =, <, >, or `` ` ``.

A single-quoted attribute value consists of `'`, zero or more characters not including `'`, and a final `'`.

A double-quoted attribute value consists of `"`, zero or more characters not including `"`, and a final `"`.

An open tag consists of a < character, a [tag name], zero or more [attributes], optional [whitespace], an optional / character, and a > character.

A closing tag consists of the string `</`, a [tag name], optional [whitespace], and the character >.

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

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

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

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

An HTML tag consists of an [open tag], a [closing tag], an [HTML comment], a [processing instruction], a [declaration], or a [CDATA section].

Here are some simple open tags:

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

Empty elements:

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

[Whitespace] is allowed:

```
<a  /><b2
data="foo" >
.
<p><a  /><b2
data="foo" ></p>
```

With attributes:

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

Custom tag names can be used:

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

Illegal tag names, not parsed as HTML:

```
<33> <__>
.
<p>&lt;33&gt; &lt;__&gt;</p>
```

Illegal attribute names:

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

Illegal attribute values:

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

Illegal [whitespace]:

```
< a><
foo><bar/ >
<foo bar=baz
bim!bop />
.
<p>&lt; a&gt;&lt;
foo&gt;&lt;bar/ &gt;
&lt;foo bar=baz
bim!bop /&gt;</p>
```

Missing [whitespace]:

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

Closing tags:

```
</a></foo >
.
<p></a></foo ></p>
```

Illegal attributes in closing tag:

```
</a href="foo">
.
<p>&lt;/a href=&quot;foo&quot;&gt;</p>
```

Comments:

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

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

Not comments:

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

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

Processing instructions:

```
foo <?php echo $a; ?>
.
<p>foo <?php echo $a; ?></p>
```

Declarations:

```
foo <!ELEMENT br EMPTY>
.
<p>foo <!ELEMENT br EMPTY></p>
```

CDATA sections:

```
foo <![CDATA[>&<]]>
.
<p>foo <![CDATA[>&<]]></p>
```

Entity and numeric character references are preserved in HTML attributes:

```
foo <a href="&ouml;">
.
<p>foo <a href="&ouml;"></p>
```

Backslash escapes do not work in HTML attributes:

```
foo <a href="\*">
.
<p>foo <a href="\*"></p>
```

```
<a href="\"">
.
<p>&lt;a href=&quot;&quot;&quot;&gt;</p>
```

6.9 Hard line breaks

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

```
foo
baz
.
<p>foo<br />
baz</p>
```

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

```
foo\
baz
.
<p>foo<br />
baz</p>
```

More than two spaces can be used:

```
foo
baz
.
<p>foo<br />
baz</p>
```

Leading spaces at the beginning of the next line are ignored:

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

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

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

```
*foo
bar*
.
```

(次のページに続く)

(前のページからの続き)

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

```
*foo\
bar*
.
<p><em>foo<br />
bar</em></p>
```

Line breaks do not occur inside code spans

```
`code
span`
.
<p><code>code  span</code></p>
```

```
`code\
span`
.
<p><code>code\ span</code></p>
```

or HTML tags:

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

```
<a href="foo\
bar">
.
<p><a href="foo\
bar"></p>
```

Hard line breaks are for separating inline content within a block. Neither syntax for hard line breaks works at the end of a paragraph or other block element:

```
foo\
.
<p>foo\</p>
```

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

```
### foo\
.
<h3>foo\</h3>
```

```
### foo
.
<h3>foo</h3>
```

6.10 Soft line breaks

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

```
foo
baz
.
<p>foo
baz</p>
```

Spaces at the end of the line and beginning of the next line are removed:

```
foo
 baz
.
<p>foo
baz</p>
```

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

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

6.11 Textual content

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

```
hello $.;'there
.
<p>hello $.;'there</p>
```

```
Foo χρ^^e1^^bf^^86ν
.
<p>Foo χρ^^e1^^bf^^86ν</p>
```

Internal spaces are preserved verbatim:

```
Multiple     spaces
.
<p>Multiple     spaces</p>
```

第 7 章

Appendix: A parsing strategy

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

7.1 Overview

Parsing has two phases:

1. In the first phase, lines of input are consumed and the block structure of the document---its division into paragraphs, block quotes, list items, and so on---is constructed. Text is assigned to these blocks but not parsed. Link reference definitions are parsed and a map of links is constructed.

2. In the second phase, the raw text contents of paragraphs and headings are parsed into sequences of Markdown inline elements (strings, code spans, links, emphasis, and so on), using the map of link references constructed in phase 1.

At each point in processing, the document is represented as a tree of **blocks**. The root of the tree is a `document` block. The `document` may have any number of other blocks as **children**. These children may, in turn, have other blocks as children. The last child of a block is normally considered **open**, meaning that subsequent lines of input can alter its contents. (Blocks that are not open are **closed**.) Here, for example, is a possible document tree, with the open blocks marked by arrows:

```
-> document
  -> block_quote
       paragraph
         "Lorem ipsum dolor\nsit amet."
    -> list (type=bullet tight=true bullet_char=-)
         list_item
           paragraph
             "Qui *quodsi iracundia*"
      -> list_item
```

(次のページに続く)

(前のページからの続き)

```
        -> paragraph
             "aliquando id"
```

7.2 Phase 1: block structure

Each line that is processed has an effect on this tree. The line is analyzed and, depending on its contents, the document may be altered in one or more of the following ways:

1. One or more open blocks may be closed.
2. One or more new blocks may be created as children of the last open block.
3. Text may be added to the last (deepest) open block remaining on the tree.

Once a line has been incorporated into the tree in this way, it can be discarded, so input can be read in a stream.

For each line, we follow this procedure:

1. First we iterate through the open blocks, starting with the root document, and descending through last children down to the last open block. Each block imposes a condition that the line must satisfy if the block is to remain open. For example, a block quote requires a > character. A paragraph requires a non-blank line. In this phase we may match all or just some of the open blocks. But we cannot close unmatched blocks yet, because we may have a [lazy continuation line].
2. Next, after consuming the continuation markers for existing blocks, we look for new block starts (e.g. > for a block quote). If we encounter a new block start, we close any blocks unmatched in step 1 before creating the new block as a child of the last matched block.
3. Finally, we look at the remainder of the line (after block markers like >, list markers, and indentation have been consumed). This is text that can be incorporated into the last open block (a paragraph, code block, heading, or raw HTML).

Setext headings are formed when we see a line of a paragraph that is a [setext heading underline].

Reference link definitions are detected when a paragraph is closed; the accumulated text lines are parsed to see if they begin with one or more reference link definitions. Any remainder becomes a normal paragraph.

We can see how this works by considering how the tree above is generated by four lines of Markdown:

```
> Lorem ipsum dolor
sit amet.
> - Qui *quodsi iracundia*
> - aliquando id
```

At the outset, our document model is just

```
-> document
```

The first line of our text,

```
> Lorem ipsum dolor
```

causes a `block_quote` block to be created as a child of our open `document` block, and a `paragraph` block as a child of the `block_quote`. Then the text is added to the last open block, the `paragraph`:

```
-> document
  -> block_quote
    -> paragraph
         "Lorem ipsum dolor"
```

The next line,

```
sit amet.
```

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

```
-> document
  -> block_quote
    -> paragraph
         "Lorem ipsum dolor\nsit amet."
```

The third line,

```
> - Qui *quodsi iracundia*
```

causes the `paragraph` block to be closed, and a new `list` block opened as a child of the `block_quote`. A `list_item` is also added as a child of the `list`, and a `paragraph` as a child of the `list_item`. The text is then added to the new `paragraph`:

```
-> document
  -> block_quote
       paragraph
         "Lorem ipsum dolor\nsit amet."
    -> list (type=bullet tight=true bullet_char=-)
      -> list_item
        -> paragraph
             "Qui *quodsi iracundia*"
```

The fourth line,

```
> - aliquando id
```

causes the `list_item` (and its child the `paragraph`) to be closed, and a new `list_item` opened up as child of the `list`. A `paragraph` is added as a child of the new `list_item`, to contain the text. We thus obtain the final tree:

```
-> document
  -> block_quote
       paragraph
         "Lorem ipsum dolor\nsit amet."
    -> list (type=bullet tight=true bullet_char=-)
         list_item
           paragraph
             "Qui *quodsi iracundia*"
      -> list_item
        -> paragraph
             "aliquando id"
```

7.3 Phase 2: inline structure

Once all of the input has been parsed, all open blocks are closed.

We then "walk the tree," visiting every node, and parse raw string contents of paragraphs and headings as inlines. At this point we have seen all the link reference definitions, so we can resolve reference links as we go.

```
document
  block_quote
    paragraph
      str "Lorem ipsum dolor"
      softbreak
      str "sit amet."
    list (type=bullet tight=true bullet_char=-)
      list_item
        paragraph
          str "Qui "
          emph
            str "quodsi iracundia"
      list_item
        paragraph
          str "aliquando id"
```

Notice how the [line ending] in the first paragraph has been parsed as a `softbreak`, and the asterisks in the first list item have become an `emph`.

7.3.1 An algorithm for parsing nested emphasis and links

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

When we're parsing inlines and we hit either

- a run of `*` or `_` characters, or
- a `[` or `![`

we insert a text node with these symbols as its literal content, and we add a pointer to this text node to the delimiter stack.

The [delimiter stack] is a doubly linked list. Each element contains a pointer to a text node, plus information about

- the type of delimiter (`[`, `![`, `*`, `_`)
- the number of delimiters,
- whether the delimiter is "active" (all are active to start), and
- whether the delimiter is a potential opener, a potential closer, or both (which depends on what sort of characters precede and follow the delimiters).

When we hit a `]` character, we call the *look for link or image* procedure (see below).

When we hit the end of the input, we call the *process emphasis* procedure (see below), with `stack_bottom` = NULL.

look for link or image

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

- If we don't find one, we return a literal text node `]`.
- If we do find one, but it's not *active*, we remove the inactive delimiter from the stack, and return a literal text node `]`.
- If we find one and it's active, then we parse ahead to see if we have an inline link/image, reference link/image, compact reference link/image, or shortcut reference link/image.
  - If we don't, then we remove the opening delimiter from the delimiter stack and return a literal text node `]`.
  - If we do, then
    - We return a link or image node whose children are the inlines after the text node pointed to by the opening delimiter.
    - We run *process emphasis* on these inlines, with the `[` opener as `stack_bottom`.

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

process emphasis

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

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

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

Then we repeat the following until we run out of potential closers:

- Move `current_position` forward in the delimiter stack (if needed) until we find the first potential closer with delimiter `*` or `_`. (This will be the potential closer closest to the beginning of the input -- the first one in parse order.)
- Now, look back in the stack (staying above `stack_bottom` and the `openers_bottom` for this delimiter type) for the first matching potential opener ("matching" means same delimiter).
- If one is found:
  - Figure out whether we have emphasis or strong emphasis: if both closer and opener spans have length >= 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.