kramdown

fast, pure-Ruby Markdown-superset converter

Contents

Support kramdown

If you like kramdown and would like to support it, you are welcome to make a small donation (PayPal or Pledgie) -- it will surely be appreciated! Thanks!

Sponsors

GROSSWEBER provides software development consulting and training services. We like to work on open source. We use it heavily. We love kramdown!

kramdown Syntax

This is version **1.9.0** of the syntax documentation.

The kramdown syntax is based on the Markdown syntax and has been enhanced with features that are found in other Markdown implementations like Maruku, PHP Markdown Extra and Pandoc. However, it strives to provide a strict syntax with definite rules and therefore isn't completely compatible with Markdown. Nonetheless, most Markdown documents should work fine when parsed with kramdown. All places where the kramdown syntax differs from the Markdown syntax are highlighted.

Following is the complete syntax definition for all elements kramdown supports. Together with the documentation on the available converters, it is clearly specified what you will get when a kramdown document is converted.

Source Text Formatting

A kramdown document may be in any encoding, for example ASCII, UTF-8 or ISO-8859-1, and the output will have the same encoding as the source.

The document consists of two types of elements, block-level elements and span-level elements:

- Block-level elements define the main structure of the content, for example, what part of the text should be a paragraph, a list, a blockquote and so on.
- Span-level elements mark up small text parts as, for example, emphasized text or a link.

Thus span-level elements can only occur inside block-level elements or other span-level elements.

You will often find references to the "first column" or "first character" of a line in a block-level element descriptions. Such a reference is always to be taken relative to the current indentation level because some block-level elements open up a new indentation level (e.g. blockquotes). The beginning of a kramdown document opens up the default indentation level which begins at the first column of the text.

Line Wrapping

Some lightweight markup syntax don't work well in environments where lines are hard-wrapped. For example, this is the case with many email programs. Therefore kramdown allows content like paragraphs or blockquotes to be hard-wrapped, i.e. broken across lines. This is sometimes referred to as "lazy syntax" since the indentation or line prefix required for the first line of content is not required for the consecutive lines.

Block-level elements that support line wrapping always end when one of the following conditions is met:

- a blank line, an EOB marker line, a block IAL or the end of the document (i.e. a block boundary),
- or an HTML block.

Line wrapping is allowed throughout a kramdown document but there are some block-level elements that do not support being hard-wrapped:

headers

> This is not an issue in most situations since headers normally fit on one line. If a header text gets too long for one line, you need to use HTML syntax instead.

fenced code blocks

> The delimiting lines of a fenced code block do not support hard-wrapping. Since everything between the delimiting lines is taken as is, the content of a fenced code block does also not support hard-wrapping.

definition list terms

> Each definition term has to appear on a separate line. Hard-wrapping would therefore introduce additional definition terms. The definitions themselves, however, do support hard-wrapping.

tables

> Since each line of a kramdown table describes one table row or a separator, it is not possible to hard-wrap tables.

Note that it is **NOT** recommended to use lazy syntax to write a kramdown document. The flexibility that the kramdown syntax offers due to the issue of line wrapping hinders readability and should therefore not be used.

Usage of Tabs

kramdown assumes that tab stops are set at multiples of four. This is especially important when using tabs for indentation in lists. Also, tabs may only be used at the beginning of a line when indenting text and must not be preceded by spaces. Otherwise the results may be unexpected.

Automatic and Manual Escaping

Depending on the output format, there are often characters that need special treatment. For example, when converting a kramdown document to HTML one needs to take care of the characters `<`, `>` and `&`. To ease working with these special characters, they are automatically and correctly escaped depending on the output format.

This means, for example, that you can just use `<`, `>` and `&` in a kramdown document and need not think about when to use their HTML entity counterparts. However, if you do use HTML entities or HTML tags which use one of the characters, the result will be correct nonetheless!

Since kramdown also uses some characters to mark-up the text, there needs to be a way to escape these special characters so that they can have their normal meaning. This can be done by using backslash escapes. For example, you can use a literal back tick like this:

```
This \`is not a code\` span!
```

Following is a list of all the characters (character sequences) that can be escaped:

```
\         backslash
.         period
*         asterisk
```

```
_          underscore
+          plus
-          minus
=          equal sign
`          back tick
()[]{}<>   left and right parens/brackets/braces/angle brackets
#          hash
!          bang
<<         left guillemet
>>         right guillemet
:          colon
|          pipe
″          double quote
’          single quote
$          dollar sign
```

Block Boundaries

Some block-level elements have to start and/or end on so called block boundaries, as stated in their documentation. There are two cases where block boundaries come into play:

- If a block-level element has to start on a block boundary, it has to be preceded by either a blank line, an EOB marker, a block IAL or it has to be the first element.
- If a block-level element has to end on a block boundary, it has to be followed by either a blank line, an EOB marker, a block IAL or it has to be the last element.

Structural Elements

All structural elements are block-level elements and they are used to structure the content. They can mark up some text as, for example, a simple paragraph, a quote or as a list item.

Blank lines

Any line that just contains white space characters such as spaces and tabs is considered a blank line by kramdown. One or more consecutive blank lines are handled as one empty blank line. Blank lines are used to separate block-level elements from each other and in this case they don’ t have semantic meaning. However, there are some cases where blank lines do have a semantic meaning:

- When used in headers – see the headers section
- When used in code blocks – see the code blocks section
- When used in lists – see the lists section
- When used in math blocks – see the math blocks section
- When used for elements that have to start/end on block boundaries

Paragraphs

Paragraphs are the most used block-level elements. One or more consecutive lines of text are interpreted as one paragraph. The first line of a paragraph may be indented up to three spaces, the other lines can have any amount of indentation because paragraphs support line wrapping. In addition to the rules outlined in the section about line wrapping, a paragraph ends when a definition list line is encountered.

You can separate two consecutive paragraphs from each other by using one or more blank lines. Notice that a line break in the source does not mean a line break in the output (due to the lazy syntax)!. If you want to have an explicit line break (i.e. a `<br />` tag) you need to end a line with two or more spaces or two backslashes! Note, however, that a line break on the last text line of a paragraph is not possible and will be ignored. Leading and trailing spaces will be stripped from the paragraph text.

The following gives you an example of how paragraphs look like:

```
This·para·line·starts·at·the·first·column.·However,
·····the·following·lines·can·be·indented·any·number·of·spaces/tabs.
···The·para·continues·here.

··This·is·another·paragraph,·not·connected·to·the·above·one.·But··
with·a·hard·line·break.·\\
And·another·one.
```

Headers

kramdown supports so called Setext style and atx style headers. Both forms can be used inside a single document.

Setext Style

Setext style headers have to start on a block boundary with a line of text (the header text) and a line with only equal signs (for a first level header) or dashes (for a second level header). The header text may be indented up to three spaces but any leading or trailing spaces are stripped from the header text. The amount of equal signs or dashes is not significant, just one is enough but more may look better. The equal signs or dashes have to begin at the first column. For example:

```
First level header
==================

Second level header
------

   Other first level header
=
```

Since Setext headers start on block boundaries, this means in most situations that they have to be preceded by a blank line. However, blank lines are not necessary after a Setext header:

```
This is a normal
paragraph.

And A Header
------------
And a paragraph

> This is a blockquote.

And A Header
------------
```

However, it is generally a good idea to also use a blank line after a Setext header because it looks more appropriate and eases reading of the document.

> The original Markdown syntax allows one to omit the blank line before a Setext header. However, this leads to ambiguities and makes reading the document harder than necessary. Therefore it is not allowed in a kramdown document.

An edge case worth mentioning is the following:

```
header
---
para
```

One might ask if this represents two paragraphs separated by a horizontal rule or a second level header and a paragraph. As suggested by the wording in the example, the latter is the case. The general rule is that Setext headers are processed before horizontal rules.

atx Style

atx style headers have to start on a block boundary with a line that contains one or more hash characters and then the header text. No spaces are allowed before the hash characters. The number of hash characters specifies the heading level: one hash character gives you a first level heading, two a second level heading and so on until the maximum of six hash characters for a sixth level heading. You may optionally use any number of hashes at the end of the line to close the header. Any leading or trailing spaces are stripped from the header text. For example:

```
# First level header

### Third level header    ###

## Second level header ######
```

> Again, the original Markdown syntax allows one to omit the blank line before an atx style header.

Specifying a Header ID

kramdown supports a nice way for explicitly setting the header ID which is taken

from PHP Markdown Extra and Maruku: If you follow the header text with an opening curly bracket (separated from the text with a least one space), a hash, the ID and a closing curly bracket, the ID is set on the header. If you use the trailing hash feature of atx style headers, the header ID has to go after the trailing hashes. For example:

```
Hello       {#id}
-----

# Hello     {#id}

# Hello #   {#id}
```

This additional syntax is not part of standard Markdown.

Blockquotes

A blockquote is started using the > marker followed by an optional space and the content of the blockquote. The marker itself may be indented up to three spaces. All following lines, whether they are started with the blockquote marker or just contain text, belong to the blockquote because blockquotes support line wrapping.

The contents of a blockquote are block-level elements. This means that if you are just using text as content that it will be wrapped in a paragraph. For example, the following gives you one blockquote with two paragraphs in it:

```
> This is a blockquote.
>    on multiple lines
that may be lazy.
>
> This is the second paragraph.
```

Since the contents of a blockquote are block-level elements, you can nest blockquotes and use other block-level elements (this is also the reason why blockquotes need to support line wrapping):

```
> This is a paragraph.
>
> > A nested blockquote.
>
> ## Headers work
>
> * lists too
>
> and all other block-level elements
```

Note that the first space character after the > marker does not count when counting spaces for the indentation of the block-level elements inside the blockquote! So code blocks will have to be indented with five spaces or one space and one tab, like this:

```
> A code block:
>
>     ruby -e 'puts :works'
```

Line wrapping allows one to be lazy but hinders readability and should therefore be avoided, especially with blockquotes. Here is an example of using blockquotes with

line wrapping:

```
> This is a paragraph inside
a blockquote.
>
> > This is a nested paragraph
that continues here
> and here
> > and here
```

Code Blocks

Code blocks can be used to represent verbatim text like markup, HTML or a program fragment because no syntax is parsed within a code block.

Standard Code Blocks

A code block can be started by using four spaces or one tab and then the text of the code block. All following lines containing text, whether they adhere to this syntax or not, belong to the code block because code blocks support line wrapping). A wrapped code line is automatically appended to the preceding code line by substituting the line break with a space character. The indentation (four spaces or one tab) is stripped from each line of the code block.

> The original Markdown syntax does not allow line wrapping in code blocks.

Note that consecutive code blocks that are only separate by blank lines are merged together into one code block:

```
    Here comes some code

    This text belongs to the same code block.
```

If you want to have one code block directly after another one, you need to use an EOB marker to separate the two:

```
    Here comes some code
^
    This one is separate.
```

Fenced Code Blocks

> This alternative syntax is not part of the original Markdown syntax. The idea and syntax comes from the PHP Markdown Extra package.

kramdown also supports an alternative syntax for code blocks which does not use indented blocks but delimiting lines. The starting line needs to begin with three or more tilde characters (~) and the closing line needs to have at least the number of tildes the starting line has. Everything between is taken literally as with the other syntax but there is no need for indenting the text. For example:

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

```
Here comes some code.
~~~~~~~~
```

If you need lines of tildes in such a code block, just start the code block with more tildes. For example:

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

code with tildes
~~~~~~~~
~~~~~~~~~~~~~~~~~~
```

This type of code block is especially useful for copy-pasted code since you don't need to indent the code.

Language of Code Blocks

You can tell kramdown the language of a code block by using an IAL:

```
~~~
def what?
  42
end
~~~
{: .language-ruby}
```

The specially named class `language-ruby` tells kramdown that this code block is written in the Ruby language. Such information can be used, for example, by converters to do syntax highlighting on the code block.

Fenced code blocks provide an easier way to specify the language, namely by appending the language of the code block to the end of the starting line:

```
~~~ ruby
def what?
  42
end
~~~
```

Lists

kramdown provides syntax elements for creating ordered and unordered lists as well as definition lists.

Ordered and Unordered lists

Both ordered and unordered lists follow the same rules.

A list is started with a list marker (in case of unordered lists one of `+`, `-` or `*` – you can mix them – and in case of ordered lists a number followed by a period) followed by one tab or at least one space, optionally followed by an IAL that should be applied to the list item and then the first part of the content of the list item. The leading tabs or spaces are stripped away from this first line of content to allow for a nice alignment

with the following content of a list item (see below). All following list items with the same marker type (unordered or ordered) are put into the same list. The numbers used for ordered lists are irrelevant, an ordered list always starts at 1.

The following gives you an unordered list and an ordered list:

```
* kram
+ down
- now

1. kram
2. down
3. now
```

> The original Markdown syntax allows the markers of ordered and unordered lists to be mixed, the first marker specifying the list type (ordered or unordered). This is not allowed in kramdown. As stated, the above example will give you two lists (an unordered and an ordered) in kramdown and only one unordered list in Markdown.

The first list marker in a list may be indented up to three spaces. The column number of the first non-space character which appears after the list item marker on the same line specifies the indentation that has to be used for the following lines of content of the list item. If there is no such character, the indentation that needs to be used is four spaces or one tab. Indented lines may be followed by lines containing text with any amount of indentation due to line wrapping. Note, however, that in addition to the rules outlined in the section about line wrapping, a list item also ends when a line with another list item marker is encountered – see the next paragraph.

The indentation is stripped from the content and the content (note that the content naturally also contains the content of the line with the item marker) is processed as text containing block-level elements. All other list markers in the list may be indented up to three spaces or the number of spaces used for the indentation of the last list item minus one, whichever number is smaller. For example:

```
* This is the first line. Since the first non-space characters appears in
  column 3, all other indented lines have to be indented 2 spaces.
However, one could be lazy and not indent a line but this is not
recommended.
*       This is the another item of the list. It uses a different number
   of spaces for indentation which is okay but should generally be avoided.
   * The list item marker is indented 3 spaces which is allowed but should
     also be avoided and starts the third list item. Note that the lazy
     line in the second list item may make you believe that this is a
     sub-list which it isn't! So avoid being lazy!
```

So, while the above is possible and creates one list with three items, it is not advised to use different (marker and list content) indents for same level list items as well as lazy indentation! It is much better to write such a list in the following way:

```
* This is the first list item bla blabla blabla blabla blabla blabla
  blabla blabla blabla blabla blabla blabla blabla blabla blabla blabla
  blabla blabla blabla bla
* This is the another item of the list. bla blabla blabla blabla blabla
```

```
  blabla blabla blabla blabla blabla blabla blabla blabla blabla blabla
```

> The original Markdown syntax also allows you to indent the marker, however, the behaviour of what happens with the list items is not clearly specified and may surprise you.
>
> Also, Markdown uses a fixed number of spaces/tabs to indent the lines that belong to a list item!

When using tabs for indenting the content of a list item, remember that tab stops occur at multiples of four for kramdown. Tabs are correctly converted to spaces for calculating the indentation. For example:

```
*   Using a tab to indent this line, the tab only counts as three spaces
    and therefore the overall indentation is four spaces.

  1.    The tab after the marker counts here as three spaces. Since the
        indentation of the marker is three spaces and the marker itself
        takes two characters, the overall indentation needed for the
        following lines is eight spaces or two tabs.
```

It is clear that you might get unexpected results if you mix tabs and spaces or if you don’t have the tab stops set to multiples of four in your editor! Therefore this should be avoided!

The content of a list item is made up of either text or block-level elements. Simple list items only contain text like in the above examples. They are not even wrapped in a paragraph tag. If the first list text is followed by one or more blank lines, it will be wrapped in a paragraph tag:

```
* kram

* down
* now
```

In the above example, the first list item text will be wrapped in a paragraph tag since it is followed by a blank line whereas the second list item contains just text. There is obviously a problem for doing this with the last list item when it contains only text. You can circumvent this by leaving a blank line after the last list item and using an EOB marker:

```
* Not wrapped in a paragraph
* Wrapped in a paragraph due to the following blank line.

* Also wrapped in a paragraph due to the
  following blank line and the EOB marker.

^
```

The text of the last list item is also wrapped in a paragraph tag if all other list items contain a proper paragraph as first element. This makes the following use case work like expected, i.e. all the list items are wrapped in paragraphs:

```
* First list item
```

```
* Second list item

* Last list item
```

> The original Markdown syntax page specifies that list items which are separated by one or more blank lines are wrapped in paragraph tags. This means that the first text will also be wrapped in a paragraph if you have block-level elements in a list which are separated by blank lines. The above rule is easy to remember and lets you exactly specify when the first list text should be wrapped in a paragraph. The idea for the above rule comes from the Pandoc package.

As seen in the examples above, blank lines between list items are allowed.

Since the content of a list item can contain block-level elements, you can do the following:

```
*   First item

    A second paragraph

    * nested list

    > blockquote

*   Second item
```

However, there is a problem when you want to have a code block immediately after a list item. You can use an EOB marker to circumvent this problem:

```
*   This is a list item.

    The second para of the list item.
^
    A code block following the list item.
```

You can have any block-level element as first element in a list item. However, as described above, the leading tabs or spaces of the line with the list item marker are stripped away. This leads to a problem when you want to have a code block as first element. The solution to this problem is the following construct:

```
*·
·····This·is·a·code·block·(indentation·needs·to·be·4(1)+4(1)
·····spaces·(tabs)).
```

Note that the list marker needs to be followed with at least one space or tab! Otherwise the line is not recognized as the start of a list item but interpreted as a paragraph containing the list marker.

If you want to have one list directly after another one (both with the same list type, i.e. ordered or unordered), you need to use an EOB marker to separate the two:

```
* List one
^
* List two
```

Since paragraphs support line wrapping, it would usually not be possible to create compact nested list, i.e. a list where the text is not wrapped in paragraphs because there is no blank line but a sub list after it:

```
*   This is just text.
    * this is a sub list item
      * this is a sub sub list item
* This is just text,
    spanning two lines
  * this is a nested list item.
```

However, this is an often used syntax and is therefore support by kramdown.

If you want to start a paragraph with something that looks like a list item marker, you need to escape it. This is done by escaping the period in an ordered list or the list item marker in an unordered list:

```
1984\. It was great
\- others say that, too!
```

As mentioned at the beginning, an optional IAL for applying attributes to a list item can be used after the list item marker:

```
* {:.cls} This item has the class "cls".
  Here continues the above paragraph.

* This is a normal list item.
```

Definition Lists

> This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the PHP Markdown Extra package.

Definition lists allow you to assign one or more definitions to one or more terms.

A definition list is started when a normal paragraph is followed by a line with a definition marker (a colon which may be optionally indented up to three spaces), then at least one tab or one space, optionally followed by an IAL that should be applied to the list item and then the first part of the definition. The line with the definition marker may optionally be separated from the preceding paragraph by a blank line. The leading tabs or spaces are stripped away from this first line of the definition to allow for a nice alignment with the following definition content. Each line of the preceding paragraph is taken to be a term and the lines separately parsed as span-level elements.

The following is a simple definition list:

```
kramdown
: A Markdown-superset converter

Maruku
:     Another Markdown-superset converter
```

The column number of the first non-space character which appears after a definition

marker on the same line specifies the indentation that has to be used for the following lines of the definition. If there is no such character, the indentation that needs to be used is four spaces or one tab. Indented lines may be followed by lines containing text with any amount of indentation due to line wrapping. Note, however, that in addition to the rules outlined in the section about line wrapping, a list item also ends when a line with another definition marker is encountered.

The indentation is stripped from the definition and it (note that the definition naturally also contains the content of the line with the definition marker) is processed as text containing block level elements. If there is more than one definition, all other definition markers for the term may be indented up to three spaces or the number of spaces used for the indentation of the last definition minus one, whichever number is smaller. For example:

```
definition term 1
definition term 2
: This is the first line. Since the first non-space characters appears in
column 3, all other lines have to be indented 2 spaces (or lazy syntax may
  be used after an indented line). This tells kramdown that the lines
  belong to the definition.
:       This is the another definition for the same term. It uses a
        different number of spaces for indentation which is okay but
        should generally be avoided.
   : The definition marker is indented 3 spaces which is allowed but
     should also be avoided.
```

So, while the above is possible and creates a definition list with two terms and three definitions for them, it is not advised to use different (definition marker and definition) indents in the same definition list as well as lazy indentation!

The definition for a term is made up of text and/or block-level elements. If a definition is not preceded by a blank line, the first part of the definition will just be text if it would be a paragraph otherwise:

```
definition term
: This definition will just be text because it would normally be a
  paragraph and the there is no preceding blank line.

  > although the definition contains other block-level elements

: This definition *will* be a paragraph since it is preceded by a
  blank line.
```

The rules about having any block-level element as first element in a list item also apply to a definition.

Tables

> This syntax feature is not part of the original Markdown syntax. The syntax is based on the one from the PHP Markdown Extra package.

Sometimes one wants to include simple tabular data in a kramdown document for which using a full-blown HTML table is just too much. kramdown supports this with a

simple syntax for ASCII tables.

Tables can be created with or without a leading pipe character: If the first line of a table contains a pipe character at the start of the line (optionally indented up to three spaces), then all leading pipe characters (i.e. pipe characters that are only preceded by whitespace) are ignored on all table lines. Otherwise they are not ignored and count when dividing a table line into table cells.

There are four different line types that can be used in a table:

- Table rows define the content of a table.

  A table row is any line that contains at least one pipe character and is not identified as any other type of table line! The table row is divided into individual table cells by pipe characters. An optional trailing pipe character is ignored. Note that literal pipe characters need to be escaped except if they occur in code spans or HTML `<code>` elements!

  Header rows, footer rows and normal rows are all done using these table rows. Table cells can only contain a single line of text, no multi-line text is supported. The text of a table cell is parsed as span-level elements.

  Here are some example table rows:

  ```
  | First cell|Second cell|Third cell
  | First | Second | Third |

  First | Second | | Fourth |
  ```

- Separator lines are used to split the table body into multiple body parts.

  A separator line is any line that contains only pipes, dashes, pluses, colons and spaces and which contains at least one dash and one pipe character. The pipe and plus characters can be used to visually separate columns although this is not needed. Multiple separator lines after another are treated as one separator line.

  Here are some example separator lines:

  ```
  |----+----|
  +----|----+
  |---------|
  |-
  | :-----: |
  -|-
  ```

- The first separator line after at least one table row is treated specially, namely as header separator line. It is used to demarcate header rows from normal table rows and/or to set column alignments. All table rows above the header separator line are considered to be header rows.

  The header separator line can be specially formatted to contain column

alignment definitions: An alignment definition consists of an optional space followed by an optional colon, one or more dashes, an optional colon and another optional space. The colons of an alignment definition are used to set the alignment of a column: if there are no colons, the column uses the default alignment, if there is a colon only before the dashes, the column is left aligned, if there are colons before and after the dashes, the column is center aligned and if there is only a colon after the dashes, the column is right aligned. Each alignment definition sets the alignment for one column, the first alignment definition for the first column, the second alignment definition for the second column and so on.

Here are some example header separator lines with alignment definitions:

```
|---+---+---|
+ :-: |:-----:| ---:|
| :-: :- -: -
:-: | :-
```

- A footer separator line is used to demarcate footer rows from normal table rows. All table rows below the footer separator line are considered to be footer rows.

  A footer separator line is like a normal separator line except that dashes are replaced by equal signs. A footer separator line may only appear once in a table. If multiple footer separator lines are used in one table, only the last is treated as footer separator line, all others are treated as normal separator lines. Normal separator lines that are used after the footer separator line are ignored.

  Here are some example footer separator lines:

```
|====+====|
+====|====+
|=========|
|=
```

Trailing spaces or tabs are ignored in all cases. To simplify table creation and maintenance, header, footer and normal separator lines need not specify the same number of columns as table rows; even `|-` and `|=` are a valid separators.

Given the above components, a table is specified by

- an optional separator line,
- optionally followed by zero, one or more table rows followed by a header separator line,
- one or more table rows, optionally interspersed with separator lines,
- optionally followed by a footer separator line and zero, one or more table rows and
- an optional trailing separator line.

Also note

- that the first line of a table must not have more than three spaces of indentation before the first non-space character,
- that each line of a table needs to have at least one not escaped pipe character so that kramdown recognizes it as a line belonging to the table and
- that tables have to start and end on block boundaries!

  The table syntax differs from the one used in PHP Markdown Extra as follows:

  - kramdown tables do not need to have a table header.
  - kramdown tables can be structured using separator lines.
  - kramdown tables can contain a table footer.
  - kramdown tables need to be separated from other block-level elements.

Here is an example for a kramdown table with a table header row, two table bodies and a table footer row:

```
|-----------------+------------+-----------------+----------------|
| Default aligned |Left aligned| Center aligned  | Right aligned  |
|-----------------|:-----------|:---------------:|---------------:|
| First body part |Second cell | Third cell      | fourth cell    |
| Second line     |foo         | **strong**      | baz            |
| Third line      |quux        | baz             | bar            |
|-----------------+------------+-----------------+----------------|
| Second body     |            |                 |                |
| 2 line          |            |                 |                |
|=================+============+=================+================|
| Footer row      |            |                 |                |
|-----------------+------------+-----------------+----------------|
```

The above example table is rather time-consuming to create without the help of an ASCII table editor. However, the table syntax is flexible and the above table could also be written like this:

```
|---
| Default aligned | Left aligned | Center aligned | Right aligned
|-|:-|:-:|-:
| First body part | Second cell | Third cell | fourth cell
| Second line |foo | **strong** | baz
| Third line |quux | baz | bar
|---
| Second body
| 2 line
|===
| Footer row
```

Horizontal Rules

A horizontal rule for visually separating content is created by using three or more asterisks, dashes or underscores (these may not be mixed on a line), optionally separated by spaces or tabs, on an otherwise blank line. The first asterisk, dash or underscore may optionally be indented up to three spaces. The following examples show different possibilities to create a horizontal rule:

```
* * *

---

  _  _  _  _

_____________
```

Math Blocks

> This syntax feature is not part of the original Markdown syntax. The idea comes from the Maruku and Pandoc packages.

kramdown has built-in support for block and span-level mathematics written in LaTeX.

A math block needs to start and end on block boundaries. It is started using two dollar signs, optionally indented up to three spaces. The math block continues until the next two dollar signs (which may be on the same line or on one of the next lines) that appear at the end of a line, i.e. they may only be followed by whitespace characters. The content of a math block has to be valid LaTeX math. It is always wrapped inside a `\begin{displaymath}...\end{displaymath}` enviroment except if it begins with a `\begin` statement.

The following kramdown fragment

```
$$
\begin{align*}
  & \phi(x,y) = \phi \left(\sum_{i=1}^n x_ie_i, \sum_{j=1}^n y_je_j \right)
  = \sum_{i=1}^n \sum_{j=1}^n x_i y_j \phi(e_i, e_j) = \\
  & (x_1, \ldots, x_n) \left( \begin{array}{ccc}
      \phi(e_1, e_1) & \cdots & \phi(e_1, e_n) \\
      \vdots & \ddots & \vdots \\
      \phi(e_n, e_1) & \cdots & \phi(e_n, e_n)
    \end{array} \right)
  \left( \begin{array}{c}
      y_1 \\
      \vdots \\
      y_n
    \end{array} \right)
\end{align*}
$$
```

renders (using Javascript library MathJax) as

$$\phi(x,y) = \phi\left(\sum_{i=1}^{n} x_i e_i, \sum_{j=1}^{n} y_j e_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} x_i y_j \phi(e_i, e_j) =$$
$$(x_1, \ldots, x_n)\begin{pmatrix} \phi(e_1, e_1) & \cdots & \phi(e_1, e_n) \\ \vdots & \ddots & \vdots \\ \phi(e_n, e_1) & \cdots & \phi(e_n, e_n) \end{pmatrix}\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

Using inline math is also easy: just surround your math content with two dollar signs,

like with a math block. If you don't want to start an inline math statement, just escape the dollar signs and they will be treated as simple dollar signs.

> **Note** that LaTeX code that uses the pipe symbol `|` in inline math statements may lead to a line being recognized as a table line. This problem can be avoided by using the `\vert` command instead of `|`!

If you have a paragraph that looks like a math block but should actually be a paragraph with just an inline math statement, you need to escape the first dollar sign:

```
The following is a math block:

$$ 5 + 5 $$

But next comes a paragraph with an inline math statement:

\$$ 5 + 5 $$
```

If you don't even want the inline math statement, escape the first two dollar signs:

```
\$\$ 5 + 5 $$
```

HTML Blocks

> The original Markdown syntax specifies that an HTML block must start at the left margin, i.e. no indentation is allowed. Also, the HTML block has to be surrounded by blank lines. Both restrictions are lifted for kramdown documents. Additionally, the original syntax does not allow you to use Markdown syntax in HTML blocks which is allowed with kramdown.

An HTML block is potentially started if a line is encountered that begins with a non-span-level HTML tag or a general XML tag (opening or closing) which may be indented up to three spaces.

The following HTML tags count as span-level HTML tags and won't start an HTML block if found at the beginning of an HTML block line:

```
a abbr acronym b big bdo br button cite code del dfn em i img input
ins kbd label option q rb rbc rp rt rtc ruby samp select small span
strong sub sup textarea tt var
```

Further parsing of a found start tag depends on the tag and in which of three possible ways its content is parsed:

- Parse as raw HTML block: If the HTML/XML tag content should be handled as raw HTML, then only HTML/XML tags are parsed from this point onwards and text is handled as raw, unparsed text until the matching end tag is found or until the end of the document. Each found tag will be parsed as raw HTML again. However, if a tag has a `markdown` attribute, this attribute controls parsing of this one tag (see below).

  Note that the parser basically supports only correct XHTML! However, there are

some exceptions. For example, attributes without values (i.e. boolean attributes) are also supported and elements without content like `<hr />` can be written as `<hr>`. If an invalid closing tag is found, it is ignored.

- Parse as block-level elements: If the HTML/XML tag content should be parsed as text containing block-level elements, the remaining text on the line will be parsed by the block-level parser as if it appears on a separate line (**Caution**: This also means that if the line consists of the start tag, text and the end tag, the end tag will not be found!). All following lines are parsed as block-level elements until an HTML block line with the matching end tag is found or until the end of the document.

- Parse as span-level elements: If the HTML/XML tag content should be parsed as text containing span level elements, then all text until the next matching end tag or until the end of the document will be the content of the tag and will later be parsed by the span-level parser. This also means that if the matching end tag is inside what appears to be a code span, it is still used!

If there is text after an end tag, it will be parsed as if it appears on a separate line except when inside a raw HTML block.

Also, if an invalid closing tag is found, it is ignored.

Note that all HTML tag and attribute names are converted to lowercase!

By default, kramdown parses all block HTML tags and all XML tags as raw HTML blocks. However, this can be configured with the `parse_block_html`. If this is set to `true`, then syntax parsing in HTML blocks is globally enabled. It is also possible to enable/disable syntax parsing on a tag per tag basis using the `markdown` attribute:

- If an HTML tag has an attribute `markdown="0"`, then the tag is parsed as raw HTML block.

- If an HTML tag has an attribute `markdown="1"`, then the default mechanism for parsing syntax in this tag is used.

- If an HTML tag has an attribute `markdown="block"`, then the content of the tag is parsed as block level elements.

- If an HTML tag has an attribute `markdown="span"`, then the content of the tag is parsed as span level elements.

The following list shows which HTML tags are parsed in which mode by default when `markdown="1"` is applied or `parse_block_html` is `true`:

Parse as raw HTML

```
script style math option textarea pre code kbd samp var
```

Also, all general XML tags are parsed as raw HTML blocks.

Parse as block-level elements

```
applet button blockquote body colgroup dd div dl fieldset form iframe li
map noscript object ol table tbody thead tfoot tr td ul
```

Parse as span-level elements

```
a abbr acronym address b bdo big cite caption code del dfn dt em
h1 h2 h3 h4 h5 h6 i ins kbd label legend optgroup p pre q rb rbc
rp rt rtc ruby samp select small span strong sub sup th tt var
```

> Remember that all span-level HTML tags like `a` or `b` do not start a HTML block! However, the above lists also include span-level HTML tags in the case the `markdown` attribute is used on a tag inside a raw HTML block.

Here is a simple example input and its HTML output with `parse_block_html` set to `false`:

```
This is a para.
<div>
Something in here.
</div>
Other para.
```

```
<p>This is a para.</p>
<div>
Something in here.
</div>
<p>Other para.</p>
```

As one can see the content of the `div` tag will be parsed as raw HTML block and left alone. However, if the `markdown="1"` attribute was used on the `div` tag, the content would be parsed as block-level elements and therefore converted to a paragraph.

You can also use several HTML tags at once:

```
<div id="content"><div id="layers"><div id="layer1">
This is some text in the `layer1` div.
</div>
This is some text in the `layers` div.
</div></div>
This is a para outside the HTML block.
```

However, remember that if the content of a tag is parsed as block-level elements, the content that appears after a start/end tag but on the same line, is processed as if it appears on a new line:

```
<div markdown="1">This is the first part of a para,
which is continued here.
</div>

<p markdown="1">This works without problems because it is parsed as
span-level elements</p>

<div markdown="1">The end tag is not found because
this line is parsed as a paragraph</div>
```

Since setting `parse_block_html` to `true` can lead to some not wanted behaviour, it is generally better to selectively enable or disable block/span-level elements parsing by

using the `markdown` attribute!

Unclosed block-level HTML tags are correctly closed at the end of the document to ensure correct nesting and invalidly used end tags are removed from the output:

```
This is a para.
<div markdown="1">
Another para.
</p>

<p>This is a para.</p>
<div>
  <p>Another para.</p>
</div>
```

The parsing of processing instructions and XML comments is also supported. The content of both, PIs and XML comments, may span multiple lines. The start of a PI/XML comment may only appear at the beginning of a line, optionally indented up to three spaces. If there is text after the end of a PI or XML comment, it will be parsed as if it appears on a separate line. kramdown syntax in PIs/XML comments is not processed:

```
This is a para.
<!-- a *comment* -->
<? a processing `instruction`
   spanning multiple lines
?> First part of para,
continues here.
```

Text Markup

These elements are all span-level elements and used inside block-level elements to markup text fragments. For example, one can easily create links or apply emphasis to certain text parts.

Note that empty span-level elements are not converted to empty HTML tags but are copied as-is to the output.

Links and Images

Three types of links are supported: automatic links, inline links and reference links.

Automatic Links

This is the easiest one to create: Just surround a web address or an email address with angle brackets and the address will be turned into a proper link. The address will be used as link target and as link text. For example:

```
Information can be found on the <http://example.com> homepage.
You can also mail me: <me.example@example.com>
```

It is not possible to specify a different link text using automatic links – use the other link types for this!

Inline Links

As the wording suggests, inline links provide all information inline in the text flow. Reference style links only provide the link text in the text flow and everything else is defined elsewhere. This also allows you to reuse link definitions.

An inline style link can be created by surrounding the link text with square brackets, followed immediately by the link URL (and an optional title in single or double quotes preceded by at least one space) in normal parentheses. For example:

```
This is [a link](http://rubyforge.org) to a page.
A [link](../test "local URI") can also have a title.
And [spaces](link with spaces.html)!
```

Notes:

- The link text is treated like normal span-level text and therefore is parsed and converted. However, if you use square brackets within the link text, you have to either properly nest them or to escape them. It is not possible to create nested links!

  The link text may also be omitted, e.g. for creating link anchors.

- The link URL has to contain properly nested parentheses if no title is specified, or the link URL must be contained in angle brackets (incorrectly nested parentheses are allowed).

- The link title may not contain its delimiters and may not be empty.

- Additional link attributes can be added by using a span IAL after the inline link, for example:

  ```
  This is a [link](http://example.com){:hreflang="de"}
  ```

Reference Links

To create a reference style link, you need to surround the link text with square brackets (as with inline links), followed by optional spaces/tabs/line breaks and then optionally followed with another set of square brackets with the link identifier in them. A link identifier may not contain a closing bracket and, when specified in a link definition, newline characters; it is also not case sensitive, line breaks and tabs are converted to spaces and multiple spaces are compressed into one. For example:

```
This is a [reference style link][linkid] to a page. And [this]
[linkid] is also a link. As is [this][] and [THIS].
```

If you don’ t specify a link identifier (i.e. only use empty square brackets) or completely omit the second pair of square brackets, the link text is converted to a

valid link identifier by removing all invalid characters and inserting spaces for line breaks. If there is a link definition found for the link identifier, a link will be created. Otherwise the text is not converted to a link.

As with inline links, additional link attributes can be added by using a span IAL after the reference link.

Link Definitions

The link definition can be put anywhere in the document. It does not appear in the output. A link definition looks like this:

```
[linkid]: http://www.example.com/ "Optional Title"
```

> Link definitions are, despite being described here, non-content block-level elements.

The link definition has the following structure:

- The link identifier in square brackets, optionally indented up to three spaces,
- then a colon and one or more optional spaces/tabs,
- then the link URL which must contain at least one non-space character, or a left angle bracket, the link URL and a right angle bracket,
- then optionally the title in single or double quotes, separated from the link URL by one or more spaces or on the next line by itself indented any number of spaces/tabs.

  The original Markdown syntax also allowed the title to be specified in parenthesis. This is not allowed for consistency with the inline title.

If you have some text that looks like a link definition but should really be a link and some text, you can escape the colon after the link identifier:

```
The next paragraph contains a link and some text.

[Room 100]\: There you should find everything you need!

[Room 100]: link_to_room_100.html
```

Although link definitions are non-content block-level elements, block IALs can be used on them to specify additional attributes for the links:

```
[linkid]: http://example.com
{:hreflang="de"}
```

Images

Images can be specified via a syntax that is similar to the one used by links. The difference is that you have to use an exclamation mark before the first square bracket and that the link text of a normal link becomes the alternative text of the image link. As with normal links, image links can be written inline or reference style. For example:

```
Here comes a ![smiley](../images/smiley.png)! And here
![too](../images/other.png 'Title text'). Or ![here].
With empty alt text ![](see.jpg)
```

The link definition for images is exactly the same as the link definition for normal links. Since additional attributes can be added via span and block IALs, it is possible, for example, to specify image width and height:

```
Here is an inline ![smiley](smiley.png){:height="36px" width="36px"}.

And here is a referenced ![smile]

[smile]: smile.png
{: height="36px" width="36px"}
```

Emphasis

kramdown supports two types of emphasis: light and strong emphasis. Text parts that are surrounded with single asterisks * or underscores _ are treated as text with light emphasis, text parts surrounded with two asterisks or underscores are treated as text with strong emphasis. Surrounded means that the starting delimiter must not be followed by a space and that the stopping delimiter must not be preceded by a space.

Here is an example for text with light and strong emphasis:

```
*some text*
_some text_
**some text**
__some text__
```

The asterisk form is also allowed within a single word:

```
This is un*believe*able! This d_oe_s not work!
```

Text can be marked up with both light and strong emphasis, possibly using different delimiters. However, it is not possible to nest strong within strong or light within light emphasized text:

```
This is a ***text with light and strong emphasis***.
This **is _emphasized_ as well**.
This *does _not_ work*.
This **does __not__ work either**.
```

If one or two asterisks or underscores are surrounded by spaces, they are treated literally. If you want to force the literal meaning of an asterisk or an underscore you can backslash-escape it:

```
This is a * literal asterisk.
These are ** two literal asterisk.
As \*are\* these!
```

Code Spans

This is the span-level equivalent of the code block element. You can markup a text part as code span by surrounding it with backticks `. For example:

```
Use `<html>` tags for this.
```

Note that all special characters in a code span are treated correctly. For example, when a code span is converted to HTML, the characters <, > and & are substituted by their respective HTML counterparts.

To include a literal backtick in a code span, you need to use two or more backticks as delimiters. You can insert one optional space after the starting and before the ending delimiter (these spaces are not used in the output). For example:

```
Here is a literal `` ` `` backtick.
And here is ``  `some`  `` text (note the two spaces so that one is left
in the output!).
```

A single backtick surrounded by spaces is treated as literal backtick. If you want to force the literal meaning of a backtick you can backslash-escape it:

```
This is a ` literal backtick.
As \`are\` these!
```

As with code blocks you can set the language of a code span by using an IAL:

```
This is a Ruby code fragment `x = Class.new`{:.language-ruby}
```

HTML Spans

HTML tags cannot only be used on the block-level but also on the span-level. Span-level HTML tags can only be used inside one block-level element, it is not possible to use a start tag in one block level element and the end tag in another. Note that only correct XHTML is supported! This means that you have to use, for example,
 instead of
 (although kramdown tries to fix such errors if possible).

By default, kramdown parses kramdown syntax inside span HTML tags. However, this behaviour can be configured with the parse_span_html option. If this is set to true, then syntax parsing in HTML spans is enabled, if it is set to false, parsing is disabled. It is also possible to enable/disable syntax parsing on a tag per tag basis using the markdown attribute:

- If an HTML tag has an attribute markdown="0", then no parsing (except parsing of HTML span tags) is done inside that HTML tag.
- If an HTML tag has an attribute markdown="1", then the content of the tag is parsed as span level elements.
- If an HTML tag has an attribute markdown="block", then a warning is issued because HTML spans cannot contain block-level elements and the attribute is ignored.
- If an HTML tag has an attribute markdown="span", then the content of the tag is

parsed as span level elements.

The content of a span-level HTML tag is normally parsed as span-level elements. Note, however, that some tags like `<script>` are not parsed, i.e. their content is not modified.

Processing instructions and XML comments can also be used (their content is not parsed). However, as with HTML tags the start and the end have to appear in the same block-level element.

Span-level PIs and span-level XML comments as well as general span-level HTML and XML tags have to be preceded by at least one non whitespace character on the same line so that kramdown correctly recognizes them as span-level element and not as block-level element. However, all span HTML tags, i.e. `a`, `em`, `b`, ..., (opening or closing) can appear at the start of a line.

Unclosed span-level HTML tags are correctly closed at the end of the span-level text to ensure correct nesting and invalidly used end tags or block HTML tags are removed from the output:

```
This is </invalid>.

This <span>is automatically closed.

<p>This is .</p>

<p>This <span>is automatically closed.</span></p>
```

Also note that one or more consecutive new line characters in an HTML span tag are replaced by a single space, for example:

```
Link: <a href="some
link">text</a>

<p>Link: <a href="some link">text</a></p>
```

Footnotes

> This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the PHP Markdown Extra package.

Footnotes in kramdown are similar to reference style links and link definitions. You need to place the footnote marker in the correct position in the text and the actual footnote content can be defined anywhere in the document.

More exactly, a footnote marker can be created by placing the footnote name in square brackets. The footnote name has to start with a caret (^), followed by a word character or a digit and then optionally followed by other word characters, digits or dashes. For example:

```
This is some text.[^1]. Other text.[^footnote].
```

Footnote markers with the same name will link to the same footnote definition. The actual naming of a footnote does not matter since the numbering of footnotes is controlled via the position of the footnote markers in the document (the first found footnote marker will get the number 1, the second new footnote marker the number 2 and so on). If there is a footnote definition found for the identifier, a footnote will be created. Otherwise the footnote marker is not converted to a footnote link. Also note that all attributes set via a span IAL are ignored for a footnote marker!

A footnote definition is used to define the content of a footnote and has the following structure:

- The footnote name in square brackets, optionally indented up to three spaces,
- then a colon and one or more optional spaces,
- then the text of the footnote
- and optionally more text on the following lines which have to follow the syntax for standard code blocks (the leading four spaces/one tab are naturally stripped from the text)

  Footnote definitions are, despite being described here, non-content block-level elements.

The whole footnote content is treated like block-level text and can therefore contain any valid block-level element (also, any block-level element can be the first element). If you want to have a code block as first element, note that all leading spaces/tabs on the first line are stripped away. Here are some example footnote definitions:

```
[^1]: Some *crazy* footnote definition.

[^footnote]:
    > Blockquotes can be in a footnote.

        as well as code blocks

    or, naturally, simple paragraphs.

[^other-note]:       no code block here (spaces are stripped away)

[^codeblock-note]:
        this is now a code block (8 spaces indentation)
```

It does not matter where you put a footnote definition in a kramdown document; the content of all referenced footnote definitions will be placed at the end of the kramdown document. Not referenced footnote definitions are ignored. If more than one footnote definitions have the same footnote name, all footnote definitions but the last are ignored.

Although footnote definitions are non-content block-level elements, block IALs can be used on them to attach attributes. How these attributes are used depends on the converter.

Abbreviations

> This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the PHP Markdown Extra package.

kramdown provides a syntax to assign the full phrase to an abbreviation. When writing the text, you don't need to do anything special. However, once you add abbreviation definitions, the abbreviations in the text get marked up automatically. Abbreviations can consist of any character except a closing bracket.

An abbreviation definition is used to define the full phrase for an abbreviation and has the following structure:

- An asterisk and the abbreviation in square brackets, optionally indented up to three spaces,
- then a colon and the full phrase of the abbreviation on one line (leading and trailing spaces are stripped from the full phrase).

Later abbreviation definitions for the same abbreviation override prior ones and it does not matter where you put an abbreviation definition in a kramdown document. Empty definitions are also allowed.

Although abbreviation definitions are non-content block-level elements, block IALs can be used on them to specify additional attributes.

Here are some examples:

```
This is some text not written in HTML but in another language!

*[another language]: It's called Markdown

*[HTML]: HyperTextMarkupLanguage
{:.mega-big}
```

> Abbreviation definitions are, despite being described here, non-content block-level elements.

Typographic Symbols

> The original Markdown syntax does not support these transformations.

kramdown converts the following plain ASCII character into their corresponding typographic symbols:

- `---` will become an em-dash (like this —)
- `--` will become an en-dash (like this –)
- `...` will become an ellipsis (like this …)
- `<<` will become a left guillemet (like this «) – an optional following space will become a non-breakable space
- `>>` will become a right guillemet (like this ») – an optional leading space will become a non-breakable space

The parser also replaces normal single ’ and double quotes ″ with “fancy quotes”. There may be times when kramdown falsely replace the quotes. If this is the case, just 'escape" the quotes and they won’t be replaced with fancy ones.

Non-content elements

This section describes the non-content elements that are used in kramdown documents, i.e. elements that don’t provide content for the document but have other uses such as separating block-level elements or attaching attributes to elements.

Three non-content block-level elements are not described here because they fit better where they are:

- link definitions
- footnote definitions
- abbreviation definition

End-Of-Block Marker

> The EOB marker is not part of the standard Markdown syntax.

The End-Of-Block (EOB) marker – a ^ as first character on an otherwise empty line – is a block level element that can be used to specify the end of a block-level element even if the block-level element, after which it is used, would continue otherwise. If there is no block-level element to end, the EOB marker is simply ignored.

You won’t find an EOB marker in most kramdown documents but sometimes it is necessary to use it to achieve the wanted results which would be impossible otherwise. However, it should only be used when absolutely necessary!

For example, the following gives you one list with two items:

```
* list item one

* list item two
```

By using an EOB marker, you can make two lists with one item each:

```
* list one
^

* list two
```

Attribute List Definitions

> This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the Maruku package.

This is an implementation of Maruku's feature for adding attributes to block and span-level elements (the naming is also taken from Maruku). This block-level element is used to define attributes which can be referenced later. The Block Inline Attribute List is used to attach attributes to a block-level element and the Span Inline Attribute List is used to attach attributes to a span-level element.

Following are some examples of attribute list definitions (ALDs) and afterwards comes the syntax explanation:

```
{:ref-name: #myid .my-class}
{:other: ref-name #id-of-other title="hallo you"}
{:test: key="value \" with quote" and other='quote brace \}'}
```

An ALD line has the following structure:

- a left brace, optionally preceded by up to three spaces,
- followed by a colon, the reference name and another colon,
- followed by attribute definitions (allowed characters are backslash-escaped closing braces or any character except a not escaped closing brace),
- followed by a closing brace and optional spaces until the end of the line.

The reference name needs to start with a word character or a digit, optionally followed by other word characters, digits or dashes.

There are four different types of attribute definitions which have to be separated by one or more spaces:

references

> This must be a valid reference name. It is used to reference an other ALD so that the attributes of the other ALD are also included in this one. The reference name is ignored when collecting the attributes if no attribute definition list with this reference name exists. For example, a simple reference looks like `id`.

key-value pairs

> A key-value pair is defined by a key name, which must follow the rules for reference names, then an equal sign and then the value in single or double quotes. If you need to use the value delimiter (a single or a double quote) inside the value, you need to escape it with a backslash. Key-value pairs can be used to specify arbitrary attributes for block or span-level elements. For example, a key-value pair looks like `key1="bef \"quoted\" aft"` or `title='This is a title'`.

ID name

> An ID name is defined by using a hash and then the identifier name which needs to start with an ASCII alphabetic character (A-Z or a-z), optionally followed by other ASCII characters, digits, dashes or colons. This is a short hand for the key-value pair `id="IDNAME"` since this is often used. The ID name specifies the unique ID of a block or span-level element. For example, an ID name looks

like `#myid`.

class names

A class name is defined by using a dot and then the class name which may contain any character except whitespace, the dot character and the hash character.

This is (almost, but not quite) a short hand for the key-value pair `class="class-name"`. Almost because it actually means that the class name should be appended to the current value of the `class` attribute. The following ALDs are all equivalent:

```
{:id: .cls1 .cls2}
{:id: class="cls1" .cls2}
{:id: class="something" class="cls1" .cls2}
{:id: class="cls1 cls2"}
```

As can be seen from the example of the class names, attributes that are defined earlier are overwritten by ones with the same name defined later.

Also, everything in the attribute definitions part that does not match one of the above four types is ignored.

If there is more than one ALD with the same reference name, the attribute definitions of all the ALDs are processed like they are defined in one ALD.

Inline Attribute Lists

These elements are used to attach attributes to another element.

Block Inline Attribute Lists

This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the Maruku package.

This block-level element is used to attach attributes to another block-level element. A block inline attribute list (block IAL) has the same structure as an ALD except that the colon/reference name/colon part is replaced by a colon. A block IAL (or two or more block IALs) has to be put directly before or after the block-level element to which the attributes should be attached. If a block IAL is directly after and before a block-level element, it is applied to preceding element. The block IAL is ignored in all other cases, for example, when the block IAL is surrounded by blank lines.

Key-value pairs of an IAL take precedence over equally named key-value pairs in referenced ALDs.

Here are some examples for block IALs:

```
A simple paragraph with an ID attribute.
{: #para-one}
```

```
> A blockquote with a title
{:title="The blockquote title"}
{: #myid}

{:.ruby}
    Some code here
```

Span Inline Attribute Lists

> This syntax feature is not part of the original Markdown syntax. The idea and syntax comes from the Maruku package.

This is a version of the block inline attribute list for span-level elements. It has the same structure as the block IAL except that leading and trailing spaces are not allowed. A span IAL (or two or more span IALs) has to be put directly after the span-level element to which it should be applied, no additional character is allowed between, otherwise it is ignored and only removed from the output.

Here are some examples for span IALs:

```
This *is*{:.underline} some `code`{:#id}{:.class}.
A [link](test.html){:rel='something'} and some **tools**{:.tools}.
```

The special span IAL `{::}` contains no attributes but doesn't generate a warning either. It can be used to separate consecutive elements that would be falsely parsed if not separated. Here is an use case:

```
This *is italic*{::}*marked*{:.special} text
```

Extensions

> This syntax feature is not part of the original Markdown syntax.

Extensions provide additional functionality but use the same syntax for it. They are available as block as well as span-level elements.

The syntax for an extension is very similar to the syntax of ALDs. Here are some examples of how to specify extensions and afterwards is the syntax definition:

```
{::comment}
This text is completely ignored by kramdown - a comment in the text.
{:/comment}

Do you see {::comment}this text{:/comment}?
{::comment}some other comment{:/}

{::options key="val" /}
```

An extension can be specified with or without a body. Therefore there exist a start and an end tag for extensions. The start tag has the following structure:

- a left brace,

- followed by two colons and the extension name,
- optionally followed by a space and attribute definitions (allowed characters are backslash-escaped closing braces or any character except a not escaped closing brace – same as with ALDs),
- followed by a slash and a right brace (in case the extension has no body) or only a right brace (in case the extension has a body).

The stop tag has the following structure:

- a left brace,
- followed by a colon and a slash,
- optionally followed by the extension name,
- followed by a right brace.

A stop tag is only needed if the extension has a body!

The above syntax can be used as is for span-level extensions. The starting and ending lines for block-level extensions are defined as:

- The starting line consists of the extension start tag, optionally preceded by up to three spaces, and followed by optional spaces until the end of the line.
- The ending line consists of the extension stop tag, optionally preceded by up to three spaces, and followed by optional spaces until the end of the line.

If no end tag can be found for an extension start tag, the start tag is treated as if it has no body. If an invalid extension stop tag is found, it is ignored. If an invalid extension name is specified the extension (and the eventually specified body) are ignored.

The following extensions can be used with kramdown:

`comment`

Treat the body text as a comment which does not show in the output.

`nomarkdown`

Don’t process the body with kramdown but output it as-is. The attribute `type` specifies which converters should output the body: if the attribute is missing, all converters should output it. Otherwise the attribute value has to be a space separated list of converter names and these converters should output the body.

`options`

Should be used without a body since the body is ignored. Is used for setting the global options for the kramdown processor (for example, to disable automatic header ID generation). Note that options that are used by the parser are immediately effective whereas all other options are not! This means, for example, that it is not possible to set converter options only for some part of a

kramdown document.

Copyright © 2009-2016 Thomas Leitner
Generated by webgen