Markua

The Markua Specification

Markua Specification

Peter Armstrong

This book is for sale at http://leanpub.com/markua

This version was published on 2015-11-25

Contents

IMPORTANT NOTE FOR LEANPUB AUTHORS: MARKUA IS NOT YET LIVE IN LEANPUB!

Markua support is NOT live in Leanpub yet for anyone but admins. Heck, Markua is not even fully-specified yet. This spec is in-progress.

If you're reading this, you are a really, *really* early adopter of Markua.

I'm publishing the Markua spec in-progress, since I believe in Lean Publishing[1]. (Lean Publishing is the act of publishing an in-progress book using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.)

Markua's specification should be complete by December 18, 2015. Markua should be live in Leanpub and in the open source version by the end of March 2016. In the meantime, what Leanpub books are written in is what they have been written in for about 5 years: Leanpub Flavoured Markdown, which is documented at this link[2].

When Markua support is live on Leanpub we will provide the option of writing either in Markua or in Leanpub Flavoured Markdown.

In the meantime, please enjoy using Leanpub and Leanpub Flavoured Markdown.

Thanks,
Peter Armstrong

[1]https://leanpub.com/lean
[2]https://leanpub.com/help/manual

The Magical Typewriter

Imagine you owned a magical typewriter.

When you used this magical typewriter, you wrote with fewer distractions. You didn't just write faster, you wrote better.

With your magical typewriter, you never worried about layout. The book formatted itself.

You could hit a key on your magical typewriter to create an ebook from your manuscript with one click.

All ebook formats would be created, and they'd all look good. You'd have PDF for computers, MOBI for Kindle, and EPUB for everywhere else. The book would look great on phones.

With your magical typewriter, you could publish your book before it was even done, and get feedback from readers all over the world. You could automatically share book updates with them. You would press one key on your magical typewriter to publish a new version, and all your readers would have it instantly.

With your magical typewriter, you could easily compare your current manuscript to any other version of your manuscript that had ever existed.

When the book was done, if you decided to make a print book, you could press a key on your magical typewriter to generate InDesign with one click. Your designer or publisher could use this as a starting point for producing a great looking print book.

With your magical typewriter, you'd only have to do one thing: Write.

Wouldn't it be great if such a magical typewriter existed?

It does. At Leanpub, we're building it.

But there's one requirement for this magical typewriter to exist: a simple, coherent, open source, free, plain text format for a book manuscript.

This simple format is what authors will write their books in, instead of Word. It will enable an ecosystem of tools to emerge around it.

This simple format will be the basis for the magical typewriter.

This simple format is called Markua.

This is its specification.

Overview

What is Markua?

Markua, pronounced “mar-coo-ah”, is a plain text format designed for the writing of books and documents.

Markua is the best way in the world to write. Markua is extremely simple and powerful. If you want to write a novel in Markua, you can learn the Markua syntax that you need to know in 30 seconds (see below). If you want to write a computer programming book or doctoral thesis in Markua, order a pizza: you can learn the additional syntax you need in less than 30 minutes.

Markua is terse. When you are writing using Markua, you are writing, not programming. Once you understand Markua’s syntax, it fades into the background.

Markua is opinionated. There is only one way to do most things. (Sometime there are two ways, however, for improved Markdown compatibility. Markua breaks its own rules.)

The Markua specification specifies both the Markua syntax and the mapping from Markua syntax to book and document concepts and to HTML.

All the Markua Syntax Needed to Write a Novel

Here’s all the Markua you need to know to write a novel:

1. Write in plain text using any text editor–or even Notepad or TextEdit.
2. Chapters, sections and paragraphs are separated by blank lines. To add a blank line, you hit the Enter key twice.
3. To make a new chapter you start a line with a pound sign (`#`), followed by a space, the name of the chapter and a blank line:
   `# The Chapter Name`
4. To make a new section you start a line with two pound signs (`##`), followed by a space, the name of the section and a blank line:
   `## The Section Name`
5. To make a new paragraph just start typing. You don’t need to indent your paragraphs, but you can if you want.
6. Surround text with one asterisk (`*`) on each side to make it `*italic*`, and with two asterisks (`**`) on each side to make it `**bold**`.
7. To add a scene break, put three asterisks on a line by themselves, with blank lines above and below them. You can have spaces between the asterisks if you want:
   `* * *`

That’s it!

Markua Processors and Mappings

Markua manuscripts can be used to automatically generate every current popular type of ebook format: PDF, EPUB, MOBI and HTML. The computer programs which do this transformation are called Markua Processors. These programs understand both Markua syntax and how to generate the various output formats.

The Markua specification defines a precise mapping from Markua to HTML and to book and document concepts. While HTML is just one of the output formats of Markua, it is also the basis of EPUB. So, the mapping to HTML is fully specified and the mapping to EPUB is mostly specified. The mapping to PDF, MOBI and other formats is not discussed, and neither are the choices for the CSS to accompany the HTML produced by Markua–all of these are implementation-dependent, in order to encourage creativity and competition in the Markua ecosystem.

Minimal Formatting

Markua provides fairly minimal formatting options. When you are writing, *formatting is procrastination*. Writing is hard enough without needing to worry about formatting.

There are two types of formatting provided in Markua:

Semantic Formatting
This is formatting which has meaning in the text, and which should not be changed by a book designer.

Basic Formatting Hints
These are formatting hints which help a Markua Processor achieve a professional looking result algorithmically. These basic formatting hints typically involve things like the positioning of figures, etc.

To emphasize how little formatting is in Markua: Markua doesn't have any concept of font sizes or margins!

Markua is not intended to be used to create the layout of a high quality print book–but it can certainly create the manuscript of that print book, and a Markua Processor can automatically generate a good looking ebook from that manuscript. Some Markua Processors like Leanpub can even generate InDesign from a Markua document as a starting point for the print book design process. This is preferable to current workflows, in which a book designer typically needs to start by undoing all the non-semantic formatting that a writer did, and then redoing it properly using different tools.

The reason that basic formatting hints are added to Markua is to enable automated production of ebooks that look good enough, especially on mobile where there are no fixed page sizes.

From Markdown to Markua

Markua is based on Markdown[3]. Markdown is a plain text format for writing text which can be transformed by Markdown processors into HTML. Markdown was created by John Gruber, with help from Aaron Swartz. Markdown was described by John Gruber as follows:

[3]http://daringfireball.net/projects/markdown/

> Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you to write using an easy-to-read, easy-to-write plain text format, then convert it to structurally valid XHTML (or HTML).

The primary reason that Markdown is a great way to write is that it was designed to be this way. The design goal of Markdown is described by Gruber as follows:

> The overriding design goal for Markdown's formatting syntax is to make it as readable as possible. The idea is that a Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. While Markdown's syntax has been influenced by several existing text-to-HTML filters, the single biggest source of inspiration for Markdown's syntax is the format of plain text email.

Markua is heavily inspired by Markdown. What Markua does is maps most Markdown syntax to book concepts, as well as adding new syntax and concepts of its own.

The most popular subset of Markdown exists virtually unchanged in Markua. For example, *all* the Markua syntax described earlier as necessary to write a novel is inherited verbatim from Markdown.

One of the design goals of Markua is for it to be the most straightforward way for a Markdown document to become a book. So, if you already write your blog posts or lecture notes in Markdown, they are probably already valid Markua. You can use any Markua Processor, such as Leanpub, to turn them into an ebook with one click. Then, as you go down the path of enhancing the manuscript and adding things which only make sense in books, this process will feel like adding, not converting. The process of going from a Markdown document into a Markua ebook is mostly a process of decorating–enhancing and reinterpreting the Markdown document appropriately. In fact, one of goals is for writers who are familiar with Markdown to feel as though they are still writing in Markdown, but that Markdown somehow *grew an understanding of book concepts*. Markua is really "Markdown for books".

I, Peter Armstrong[4], am creating Markua. I'm the co-founder of Leanpub. (Unlike most specifications, this specification is being written by one person, so throughout it I will say "I", not "we". I'm not the Queen, so I shouldn't use the royal "we"!) At Leanpub, my cofounder Scott Patten[5] and I created a dialect of Markdown called "Leanpub Flavoured Markdown". Leanpub Flavoured Markdown was basically Markdown minus inline HTML plus a number of extensions needed to write books. This Markdown dialect has been used for years by Leanpub authors to create all kinds of books.

Markua is the evolution of how Markdown is used at Leanpub. After years of experience with Leanpub Flavoured Markdown, I set out to specify it properly. I originally thought that this is what Markua would be: just a formalization of Leanpub Flavoured Markdown. But as the specification work evolved, I realized that Markua was really a descendant of Markdown, not just a flavour of it.

Books and documents have different abstractions than HTML. All abstractions leak[6], so HTML's abstractions leak into Markdown and the abstractions of books and documents leak into Markua.

So, I started over.

[4]https://twitter.com/peterarmstrong
[5]https://twitter.com/scott_patten
[6]http://www.joelonsoftware.com/articles/LeakyAbstractions.html

I started with Markdown. I then removed or changed the few things which were either not needed for books and documents, or which were overly complex. I then found new, more appropriate abstractions for books and documents–most importantly, resources, figures and attribute lists. I then reassembled the pieces around these abstractions into this whole. Hopefully it is a coherent whole.

Why the Name "Markua"?

When I set out to specify Markua, I realized I needed a name. I wanted a name that conveyed the warmth and love that I have for Markdown, for writing and for writers, while not implying endorsement by John Gruber in any way. I also did not want a name which refenced Leanpub: Markua is a standalone specification with its own identity, which anyone (including Leanpub competitors) can freely implement. Finally, I was on vacation in Hawaii I named Markua, and I wanted something that sounded happy, friendly and almost Hawaiian. (Yes, I'm aware that there is no r in Hawaiian.) I also wanted a name that had its .com domain name available, and that was short and spellable, for branding purposes. The Markua name does have all these these properties, and I feel it is a really good brand for what I'm trying to accomplish.

Why is Inline HTML Not Supported in Markua?

Inline HTML is not supported for both implementation and philosophical reasons.

Supporting inline HTML would be very hard to implement, as Markua and Markdown have different use cases. Markdown is a better way to write HTML; Markua is a better way to write books and documentation. Since Markdown's only output format target is HTML, there is a strong temptation to support inline HTML–generating HTML from HTML is as simple as passing the HTML through. From an implementation perspective, Markdown gets inline HTML support for free. On the other hand, for Markua, there is a large amount of HTML that makes no sense in books and documentation. Supporting inline HTML would put an undue burden on Markua Processors, the book and document production systems which produce PDF, EPUB, MOBI and HTML from Markua documents. Mapping all of HTML to PDF or MOBI formats in a meaningful way would be difficult and arbitrary.

Philosophically, Markua is intended to be a better way to write books and documentation. By not supporting inline HTML, Markua imposes more constraints on writers who would be tempted to use inline HTML for layout purposes. Since Markua does not support inline HTML, attempting to do complex layout in Markua is just not possible. And since it's not possible, the temptation to procrastinate by doing complex layout is not there.

Supporting inline HTML would make Markua a worse format, both to write in and to implement support for. So, it's not supported.

One design consequence of the support for inline HTML in Markdown is that Markdown's syntax can stay artificially small–since Markdown authors can always fall back to using HTML directly, Markdown does not need to be able to produce all of HTML. Since Markua does not support inline HTML, any required output must be produceable by a Markua input. For example, there is no official table syntax in Markdown, and Markdown authors can just use inline HTML tables. Since Markua does not support inline HTML, and since books and documentation often require tables, Markua needs to add a table syntax.

About This Document

This is an in-progress, alpha quality specification. It is being constantly updated at https://leanpub.com/markua. To read the specification in PDF, EPUB, MOBI or HTML format, go there.

To read the specification manuscript in Markua, see https://github.com/markuadoc/markua. (Until Markua is fully supported in Leanpub, the Markua spec is a strange hybrid of Markua and Leanpub Flavoured Markdown.)

To stay current on Markua progress, follow @markuadoc[7] or @peterarmstrong[8] on Twitter.

[7]https://twitter.com/markuadoc
[8]https://twitter.com/peterarmstrong

Markua Syntax

This chapter is an informal, easy to read specification of all of Markua's syntax. In this chapter, the HTML that is generated is only briefly mentioned where it's interesting. To learn the exact mapping to HTML, and for a more formal specification, read the remaining chapters.

Markua Character Encoding

Markua documents are written in plain text.

Specifically, Markua documents must be saved in the **UTF-8** character encoding. ASCII text (what "plain text" means to most westerners) is a valid subset of UTF-8, so for many western authors their favorite text editor already saves files as UTF-8 text.

Text Formatting

Markua's goal is to provide all the semantic formatting required by authors. Formatting that could be done by a book designer without needing to discuss it with the author is considered orthogonal to Markua, and is largely omitted from Markua.

Markua is a plain text format designed for the writing of books and documents. Books and documents have various types of text formatting in them: bold, italic, underline, strikethrough, superscript and subscript.

Here's how to do basic, semantic text formatting.

Italic
: To produce *italic text*, surround it with either `_one underscore_` (producing `<i>` in HTML) or `*one asterisk*` (producing `<em>` in HTML[9]).

Bold
: To produce **bold text**, surround it with either `__two underscores__` (producing `<b>` in HTML) or `**two asterisks**` (producing `<strong>` in HTML).

Bold + Italic
: To produce ***bold + italic text***, surround it with either `___three underscores___` (producing `<b><i>` in HTML), `***three asterisks***` (producing `<strong><em>` in HTML).

Underline
: To produce underlined text, surround it with `____four underscores____` (producing `<u>` in HTML). This is gross, but it's a tradeoff for Markdown compatibility: the one, two and three underscore choices were taken. Thankfully, it's usually preferable to use italic instead of underline. However, underline is not just a typewriter version of italics. In some languages, underlining serves a distinct, legitimate purpose.

[9]To learn about the hilarious issues around bold and italic text in HTML and why there are two ways of producing bold and italic text in Markua, see the Appendix The W3C and Semantic Emphasis.

Strikethrough

To produce *strikethrough text*, surround it with `~~two tildes~~`. This is the same syntax as is used by both GitHub Flavored Markdown[10] and by John Macfarlane's excellent pandoc[11].)

TODO_LEANPUB - add strikethrough to Leanpub

Superscript

To produce superscript like the 3 in $5^3 = 125$, surround it with carets like `5^3^ = 125`. (This is the same syntax as is used by pandoc.)

Subscript

To produce subscript like the 2 in H_2O, surround it with single tildes like `H~2~O`.

Note that if the asterisks and underscores needed to produce bold and italic are not nested correctly, they will be output as asterisks and underscores instead of producing formatting. This includes combinations like `**_this**_`, `*__this*__` or `__*this__*`.

Headings

Markdown is a way of writing HTML, and HTML has 6 heading levels: `h1`, `h2`, `h3`, `h4`, `h5` and `h6`.

Markua, however, is a way of writing books and documents. Books have things like chapters, sections and subsections. Sometimes books have parts. Documents have sections and subsections.

What Markua does is to use the heading levels that Markdown uses to create HTML to create the structure of the book or document, as well as generating the `h1` through `h6` heading levels in the HTML version of the book.

Markua is also a lot more strict about heading syntax than Markdown. In Markdown, there are two syntaxes for making headings, atx and Setext, and both of them can take many forms[12]. However, **in Markua there is exactly one way of making every type of heading.** It is a subset of the atx headings in Markdown.

Here's the only way that headings are made in Markua:

[10]https://help.github.com/articles/github-flavored-markdown/
[11]http://johnmacfarlane.net/pandoc/README.html
[12]http://daringfireball.net/projects/markdown/syntax#header

```
# Part (h1) #

This is a paragraph.

# Chapter (also h1)

This is a paragraph.

## Section (h2)

This is a paragraph.

### Sub-Section (h3)

This is a paragraph.

#### Sub-Sub-Section (h4)

This is a paragraph.

##### Sub-Sub-Sub-Section (h5)

This is a paragraph.

###### Sub-Sub-Sub-Sub-Section (h6)

This is a paragraph.
```

Specificaly, to create a Markua heading, you put the following on a line by itself, with a blank line above and below it:

- between one and six pound signs (`#`)
- followed by exactly one space
- followed by text
- followed (only for parts) by exactly one space and exactly one pound sign

Making these headings divides your book or document into these divisions. If you add part headings, your book will have parts; if you don't, it won't. If you add chapter headings, your book will have chapters; if you don't, it won't.

Note that in Markua, both parts and chapters produce an `h1`. This is mildly surprising, but entirely appropriate: For books which have no parts, the chapter is the topmost organizational unit. In a book with parts, it is more important for chapters to behave the same way that they do in a book without parts (i.e. to be created with `# Title` and to produce an `h1`) than to be correctly "contained" (if you think like an outliner) inside the Part.

Markua is more strict about spaces and pound signs than Markdown:

1. In Markua, to make a part heading you make a heading like `# Title #` – the heading starts with exactly one a pound sign and exactly one space, and ends with exactly one space and exactly one pound sign. No other combination of pound signs and spaces makes a part heading.

2. Part headings end with a space and a pound sign, but other than that, headings cannot have optional pound signs at the end of them. Any other pound signs are considered to be text in the heading. (This is different than the atx headers in Markdown.)
3. All headings must start with between 1 and 6 pound signs (`#`) and then have exactly one space, followed by text. If any other number of spaces is used or if tabs are used, the text must be interpreted as a paragraph by a Markua Processor, not as a heading. (This ensures that Markua manuscripts look consistent for authors, and can be parsed more easily by Markua Processors.)
4. Headings must be separated from other block elements by blank lines both before and after them. Because of the way that Markua concatenates files, headings at the top of a Markua document automatically have a blank line above them, and headings at the bottom of a Markua document automatically have a blank line below them. So, headings at the top of a document only need a blank line below them, and headings at the bottom of a document only need a blank line above them.

I thought long and hard about Markua headings, and changed my mind (as Scott and Braden will attest) a number of times. I settled on this design when I realized that the design requirements were as follows:

1. From looking at a heading, an author must be able to know exactly what it is. It's not acceptable to need to reference metadata to know what the headings mean.
2. Only existing Markdown syntax can be used, to ensure that Markdown-aware tooling will work unchanged.
3. The number of leading pound signs must match the level of heading in HTML.
4. For a book with chapters only (no parts), the chapter headings must feel "top-level".
5. The syntax to create parts must be as economical as possible.
6. There must be only one way to create every level of heading. Headings structure Markua documents, and must be clear and unambiguous.
7. It should be possible to switch back and forth between having parts in your book without having to modify all existing headings.
8. Books with parts and chapters must be able to have the same number of levels of sub-sections.
9. Documents can have parts, chapters and sections just like books, and it's up to the document designer to create the appropriate CSS.

The design of Markua headings meets these requirements, in what I feel is an optimal way.

Leanpub Flavoured Markdown violated a number of these requirements, especially #2 (as parts were created with `-# Title`). Previous versions of Markua were even worse, violating many of these requirements (e.g. #1, #3, #4, #6, #7 and #8).

There are two primary issues with Setext headings in Markdown. First, it is inconsistent: you can only define `h1` and `h2`; to define h3 and below, the atx style of heading must be used. Second, it is confusing: it's unclear about how many equals or minus signs you need to use. (The answer is `1`, which looks disgusting.) Also, it's unclear whether you need to add a blank line below the row of equals signs or minus signs, and whether a heading is still produced if this is not done.

Whitespace, Paragraphs and Blank Lines

Together, newlines, spaces and tabs are called whitespace. By a newline, I mean the character which is produced by the enter key on the keyboard. Two newlines in a row produce a blank line, i.e. a line with no text or whitespace on it.

Paragraphs and blank lines are handled the same way in Markua and Markdown. Normal Markua text is in paragraphs. A paragraph produces a `<p>` tag in HTML. The distinguishing thing about a paragraph is that there is nothing else distinguishing about it: unlike headings, lists and other Markua elements, a paragraph requires no special formatting. To create a new paragraph, you add two newlines to create a blank line. Any extra number of newlines are ignored: two newlines in a row is equivalent to three (or ten!) newlines in a row.

Here's an example of this in Markua syntax. Throughout this specification, examples of Markua syntax will be shown as a figure which is a code resource, so that the exact Markua manuscript text can be shown unprocessed by a Markua. These are three paragraphs in Markua, each separated by the canonical two newlines which produce one blank line:

```
I'm paragraph one. Yay!

This is paragraph two.

This is paragraph three.
```

A paragraph is the least-specialized example of something called a block element. Broadly speaking, Markua documents consist of three things: block elements, span elements and metadata. All block elements that are separated by at least one blank line from other block elements, and that are not some specialized type of block element (like a heading, figure or list) are paragraphs.

Newlines, Indentation and Spaces

The goal for newline and indentation handling in Markua is for everything to just work. Prose, poetry, code and figures should just work. There should be no surprises. Above all, no invisible formatting characters can be used.

In this regard, Markua has broader design goals than Markdown: Markdown is for web writers; Markua is for *writers*. Markdown is a format for authors to write HTML; Markua is a format for authors to write books and documents, of which HTML is just one format. The concepts in Markdown are HTML concepts; the concepts in Markua are book and document concepts.

Because of this, the way that single newlines and leading spaces are handled in Markua is different from the way that they are handled in Markdown.

Single Newlines

In Markdown, a single newline inside a paragraph is interpreted as a single space. To add a forced line break (a `<br/>` tag in HTML), Markdown uses a horrible hack: you need to add two spaces at the end of the line, followed by a single newline. This means that it is **impossible** to look at a Markdown document with single newlines in it and understand what they mean: you need to find out if there are invisible formatting characters at the end of the line! Worse, in some text editors (like Emacs, the editor I use), trailing spaces at the end of a line are automatically removed when a file is saved. So, it's possible for me to inadvertently modify the meaning of a Markdown document by simply opening it and saving it! Also, in Markdown, it's impossible to type a sonnet without relying on the two space hack at the end of every line to force a newline–which isn't very poetic at all.

In Markua, a single newline inside a paragraph forces a line break, which produces a `<br/>` tag in HTML. This is true without any invisible formatting nonsense. As you will see, there are many types of block elements in Markua, including paragraphs, asides, blurbs, callouts, blockquotes and headings. A single newline inside most of these block elements produces a forced line break, and what follows is either still part of the same block element or is contained in the block element. However, headings (discussed later) are block elements, but inside a heading, a single newline is interpreted as a single space by Markua. This is because a heading is intended to be all on one line.

Leading Spaces (Prose vs. Poetry)

In Markua, leading spaces are interpreted based on context: specifically, based on what came before them.

Following **exactly one** newline, whitespace is **preserved**. Specifically, a single space produces a single space (a "non-breaking" space, or `&nbsp;`, in HTML), and a single tab produces four spaces (four "non-breaking" spaces, or `&nbsp;&nbsp;&nbsp;&nbsp;`, in HTML).

Following **two or more** newlines (one or more blank lines), whitespace is **ignored**. So, you can manually indent your paragraphs if you're used to doing so, and it will have no effect.

This distinction also helps Markua support prose and poetry without a bunch of special syntax.

Two (or more) newlines in Markua create a new paragraph in prose, or a new stanza in poetry. That's it. Paragraphs can either be indented or not indented, but this is a global setting at the book or document level–not something that is specific to a given paragraph. Authors may choose to indent their paragraphs in their manuscript based on personal preference or habit, but this should have no effect on whether the paragraphs are indented in the output.

One newline, on the other hand, is used to add a forced line break, but to keep the existing paragraph (or stanza). Because of this, any leading space after a single newline is not paragraph indentation, but is instead deliberate indentation on the specific line. As such, it should be respected.

This means you can write a sonnet without typing any special formatting characters. In the following Markua text, every line break will be added correctly, and the last two lines will be indented:

```
I grant I never saw a goddess go;
My mistress when she walks treads on the ground.
    And yet, by heaven, I think my love as rare
    As any she belied with false compare.
```

In Markdown, not only would you lose the indents on the third and fourth lines, but without any trailing space hack used this would all be collapsed to one line! (Note how this shows that the two space hack is unacceptable: you should be able to tell what output is produced by looking at the input.)

More generally, the rules about single and double newlines mean that in Markua you can just write poetry without any formatting characters. Stanzas are separated with a blank line, and any indentation is preserved except on the first line of a stanza.

Trailing Spaces are Stripped

Unlike Markdown, all trailing spaces at the end of a line are ignored by Markua. This way, there is no reliance on invisible formatting characters, and editors which strip trailing spaces have no effect on a Markua document.

Internal Spaces are Collapsed to One Space

Like Markdown, in Markua, spaces or tabs that are in the middle of a line of text are collapsed to a single space. In Markdown, the main reason for this is presumably that this is how things work in HTML (without using `&nbsp;`). In Markdown, the reasons for this are that it's consistent with Markdown, and that it lets authors write books using either one or two spaces after a period, with there being no difference in the formatting.

Because of this, if you want to write poetry like e e cummings in which the exact spacing is preserved in leading and internal spaces, you need to use a code resource. Markua defines many different types of resources, including math and code resources. Resources are discussed later in this chapter.

One Blank Line is Added When Concatenating Manuscript Files

A Markua document can be written in one file or multiple manuscript files. If a manuscript is written in multiple files, these files are concatenated together by the Markua processor to produce one temporary manuscript file, and that one file is used as the input. Importantly, in order to avoid a number of bugs, the files are not just concatenated together unchanged–they **must** be concatenated together by Markua Processors with **two newlines** added between the end of each file and the beginning of the next file. This is needed in order to separate the content of the two files with one blank line between them, in order to prevent a number of surprises for authors. Note that because of this rule, a paragraph (or any other block element) cannot span multiple manuscript files.

To see why this rule is important, consider the following single-file Markua manuscript:

⚠ TODO_LEANPUB - add caption support to Leanpub

```
# Chapter One

Lorem ipsum dolor.

# Chapter Two

Yada yada yada.
```

Suppose instead a multiple-file approach was used, in which there were two files, ch1.txt and ch2.txt, with the following content:

```
# Chapter One

Lorem ipsum dolor.
```

```
# Chapter Two

Yada yada yada.
```

If Markua did not add any newlines between files, then these files would produce the following manuscript:

```
# Chapter One

Lorem ipsum dolor.# Chapter Two

Yada yada yada.
```

If Markua only added one newline when concatenating, this would produce the following manuscript:

```
# Chapter One

Lorem ipsum dolor.
# Chapter Two

Yada yada yada.
```

However, since Markua requires that headings be separated by blank lines above and below them, the `# Chapter Two` heading would not be a heading. It would be considered part of the previous paragraph! This would be very surprising and the source of a number of bugs. Worse, since a number of text editors such as Emacs have a "strip blank lines at the end of files" setting, it would be possible to introduce such a bug if Markua simply relied on blank lines being added to the end of a file by the author.

So, because of the blank line rule, concatenating the files produces the same manuscript as the single-file manuscript above:

```
# Chapter One

Lorem ipsum dolor.

# Chapter Two

Yada yada yada.
```

Resources

Markua books and documents are written in plain text, either in one text file or multiple text files. However, modern books and documents are not just text. Books and documents embed many types of *resources.* These resources have traditionally included things like images, computer code listings and mathematical equations. More recently, with the development of ebooks and formats like EPUB3, ebooks have gained the ability to embed audio and video resources. Markua's goal is to make inserting all types of resources simple and consistent, while staying as close to Markdown's syntax as possible.

Resources vary in four different ways:

1. **Insertion Methods**: Span and Figure
2. **Locations**: Local, Web and Inline
3. **Types**: image, video, audio, code or math
4. **Formats**: `png`, `m4a`, `mp3`, `ruby`, `latex`, `plain`, etc.

There are five types of resources: image, video, audio, code and math. Each type of resource has a number of supported formats. Any of the five resource types can be inserted as a local resource or web resource, and many of the resource types can also be inserted as an inline resource.

Before going into more detail, it's helpful to consider some brief example of Markua text which shows a number of resources being inserted. These resources have different insertion methods (span and figure), different locations (local, web and inline), different types (image, code) and different formats (png, jpg, ruby)–yet the syntax to insert them is compact and consistent:

~~~
Inserting a span image is as easy as `![pie](pie.png)`. Inserting an image as a figure is also...

![A Piece of Cake](http://markua.com/cake.jpg)

Inline code resources are added as spans like `hello world` or as figures:

```ruby
puts "hello world"
```

{format: ruby}
![Hello, World](hello.rb)
~~~

The last figure in the example above showed an attribute list, which is a list of key-value pairs in curly braces. Any figure can have an attribute list, regardless of resource location, type or format.

If you're familiar with Markdown syntax, you'll note that the syntax for local and web resources is similar to Markdown's inline image insertion syntax, and that the syntax for inline resources is similar to the fenced code blocks syntax from popular Markdown extensions including GitHub Flavoured Markdown.

> Markdown's reference style image syntax is not supported in Markua for *any* type of resource–including images.
>
> There are two main reasons for not supporting reference style resource syntax in Markua:
>
> 1. Resources in Markua are complex enough without having two syntaxes.
> 2. In something as large as a book, the potential for id collisions in link definitions using the reference link syntax is a lot higher than in the blog post length of typical Markdown documents.

Resource Insertion Methods

Resources can be inserted either as spans or as figures.

Spans

When a resource is inserted as a span, the resource is inserted as part of the flow of text of a paragraph with no newlines before or after it. A span resource cannot have an attribute list.

The syntax for a local resource or a web resource inserted as a span is as follows:

```
It's just ![optional alt text](resource_path_or_url) right in the text.
```

The optional alt text is discussed shortly.

The syntax for an inline resource inserted as a span is as follows:

```
It's a single backtick `followed by inline resource content\`optional_format and then a single backtick.
```

Figures

When a resource is inserted as a figure, the resource is inserted with at least one newline before and after it. A figure can have an attribute list. A figure can either be top-level (with a blank line before and after it), or it can be part of the flow of text of a paragraph (with a single newline before it, and one or more newlines after it). The ways that figures and other block elements can be inserted in paragraphs is discussed in depth later.

The syntax for a local resource or a web resource inserted as a figure is as follows:

```
{key: value, comma: separated, optional: attribute_list}
![Optional Figure Caption](resource_path_or_url)
```

Note that if there is a figure caption specified in the attribute list and in the square brackets, the one in the attribute list wins. In the following figure, the caption would be "Foo" not "bar":

```
{caption: Foo}
![bar](foo.png)
```

The syntax for an inline resource inserted as a figure is as follows:

~~~
{key: value, comma: separated, optional: attribute_list}
```optional_format
inline resource content
```
~~~

Note that if there is a format specified in the attribute list and after the backticks, the one in the attribute list wins. In the following figure, the format is `text` not `ruby`:

~~~
{format: text}
```ruby
puts "hello world"
```
~~~

A figure can also have attributes. The supported attributes vary based on the type of resource, but all figures support the `format` and `caption` attributes.

`format`
This is the resource format. Different resource types have different legal values for format.

`caption`
This is text which is shown near the figure, typically above or below it.

Many resource types can also have alt text. Alt text is not the same thing as a figure caption.

Figure Captions

A resource which is inserted as a figure can have a caption.

This caption shows up in two places:

1. Near the resource, typically above or below it, per the preference of the Markua Processor.
2. As the name of the Figure in the List of Illustrations or Table of Figures, if one is generated for the book. In this case, the caption name should serve as the text, and be a crosslink to the caption associated with the figure itself.

Alt Text and Figure Captions

Some resources support alt text. Alt text is text which is intended to take the place of the resource if the resource itself cannot be seen. In the case of images, the obvious use case is for readers with visual disabilities who are using a screen reader, but it also includes audiobooks and ebook readers which often do not support embedded images, audio and video.

The four types of resource which can have alt text are image, video, audio and math. Resources of type code are themselves just text, so it makes no sense for them to have alt text. If any alt text is provided for a code resource it is ignored.

The alt text should **not** be the same thing as the figure caption, if the figure caption is present. (Imagine the annoyance of having a visual disability and having your screen reader read identical alt text and figure captions to you throughout an entire book!) Instead, the alt text should be descriptive of the image content, while the figure caption can be more creative. For example, a figure caption may be “Washington Crossing the Delaware” and the alt text could be “Denzel Washington riding a jet ski in a river”. Having good alt text would enable readers who cannot see the image to still get the joke which the figure caption makes.

The syntax to provide alt text varies based on the resource location and insertion method.

Span + (Local or Web)

The alt text for a local or web resource inserted as a span is provided in square brackets before the resource path or URL:

```
It's just ![optional alt text](resource_path_or_url) right in the text.
```

Here’s an example:

```
This needs to ![a stop sign](stop.png) now.
```

If you omit the optional alt text with a local or web resource, you still need to type the square brackets:

```
This needs to ![](stop.png) now.
```

Span + Inline

Alt text is not supported for inline span resources. There’s no good syntax, and almost no real use case (with the exception of an inline span SVG image). If you want alt text in a span resource, use a local or web resource instead.

Figure

The alt text for a resource inserted as a figure is provided in the attribute list above the figure. This is true regardless of whether the resource location is local, web or inline, as shown in the following examples:

~~~
{alt: "The Delaware state map"}
![Where To Incorporate](delaware.svg)

{alt: "Denzel Washington on a jet ski in a river"}
![Washington Crossing the Delaware](delaware.jpg)

{alt: "Denzel Washington on a jet ski in a river", caption: "Washington Crossing the Delaware"}
![](delaware.jpg)

{caption: "Earth From Space (Simplified)", alt: "a blue circle"}
```svg
<svg width="20" height="20">
  <circle cx="10" cy="10" r="9" fill="blue"/>
</svg>
~~~

Note that as shown above, the caption for a figure can be provided either in the metadata or in the square brackets in front of a local or web resource.

Context-Sensitive Meaning of Square Bracketed Text

You may have been surprised reading this section. Markua does not have many inconsistencies, but this is one of them: the optional text inside the square brackets with a local or web resource varies based on whether the resource is inserted as a span resource or as a figure!

Insertion Method	Text in Square Brackets is
Span	The Alt Text
Figure	The Figure Caption

The reason for this inconsistency is that even though it's a bit harder to learn, it's much better for writers this way once they've learned it.

The square brackets contain the text which is the most common use case for both span resources and for figures. For a span resource, that is the alt text. For a figure, that is the figure caption.

This is especially true since a span image cannot have a figure caption, and since not all figures are of resource types that can have alt text. (Code resources are themselves text, so they cannot have alt text.) Also, many image, audio and video resources are inserted without alt text.

So, given this rule, a figure which is an external code sample can be inserted as either:

```
![My Amazing Algorithm](algorithm.rb)

{caption: "My Amazing Algorithm}
![](algorithm.rb)
```

The first choice is clearly shorter, as well as more pleasant to write and to read. With the square bracketed text being the figure caption for figures, both ways are supported. If the square-bracketed text was alt text in figures, then only the longer, unpleasant second choice would be supported.

Note that the second choice is how you add captions to figures which are inline resources, since there are no square brackets to use:

~~~
Here's a paragraph before the figure.

{caption: "My Amazing Algorithm}
```ruby
puts "hello world"
```

Here's a paragraph after the figure.
~~~

Resource Locations

A resource is either considered a local, web or inline resource based on its location:

Local Resource
: The resource is stored along with the manuscript–either in a `resources` directory on a local filesystem, or uploaded to the same web service where the manuscript is being written.

Web Resource
: The resource is referred to via an `http` or `https` URL.

Inline Resource
: The resource is defined right in the body of a Markua document.

Local Resources

If local resources are used, all local resources must be stored inside a `resources` directory, or one of its subdirectories. The `resources` directory is not part of the path to the resource. Implementors of Markua Processors must ensure they do not support navigating upward with `../` in paths.

A file called `foo.jpg` in the `resources` directory is referenced as `![](foo.jpg)`–not as `![](/foo.jpg)`, `![](resources/foo.jpg)` or `![](/resources/foo.jpg)`.

A file called `bar.png` in a subdirectory `images` of the resources directory is referenced as `![](images/bar.png)`–not as `![](/images/bar.png)`, `![](resources/images/bar.png)` or `![](/resources/images/bar.png)`.

Markua does not specify whether there are any subdirectories of the `resources` directory, or what their names are. Since any subdirectories have their names as part of the path to the resource, implementations can do whatever they want. For example, Leanpub will create subdirectories of the `resources` directory for every type of resource (images, audio, video, code and math), but this is not a requirement.

TODO_LEANPUB - implement the resources directory support

The local resources approach can also be used by hosted services. Internally, services can store resources wherever they want, but if they provide a download (say as a zip file) they should create the resources directory and provide the uploaded resources in that directory. If a nested structure is used, it should be exported that way–if a web service produces paths which reference images inside an images directory (e.g. as `images/foo.png`), then the zip file centaining an export should contain a `resources` directory which contains an `images` subdirectory with the images.

Web Resources

If web resources are supported, both `http:` and `https:` resources should be supported.

Web resources are identified by URL. The `resource_path_or_url` is either the relative path to the resource inside the `resources` directory or the absolute URL of the resource on the internet. In the case of image, audio and video resources, this becomes the `src` attribute of the resource in HTML.

Inline Resources

Inline resources can be of type code or math (regardless of format) and they can also be image resources of `svg` format. Since an SVG image is just XML text, it can be contained in the text of a Markua document. This is not something that is true for binary resources like PNG or JPEG images or any type of audio or video file–these can only be local or web resources.

Resource Types and Formats

There are five types of resources: image, video, audio, code, and math. Each type of resource has a number of supported formats. The format determines the type.

The formats can either be specified by the `format` attribute or (in most cases) inferred from the file extension for local and web resources. (Inline resources obviously have no file extension, since they are contained in the body of a Markua manuscript file.)

Certain file extensions must be associated with a specific type of resource. These are specified in the sections below. Also, as discussed in the code section, Markua Processors must interpret all unspecified file extensions as specifying a resource of type code with a format of `guess`.

When a local or web resource is inserted as a span, the file extension is the only way to provide the format. When a local or web resource is inserted as a figure, the default type and format can be overridden by a type and/or format specified in the attribute list.

Format Detected	**Resource Type Inferred**
`gif`, `png`, `jpeg`, `svg`, `svgz`	image
`mp4`, `webm`	video
`mp3`, `aac`, `wav`, `ogg`	audio
`latex`, `mathml`	math
`text`, `guess` (unrecognized)	code

As an author, all you do is provide the correct file extension or set the format in the attribute list. Markua recognizes the format, and uses it to look up the type. (If the format is unrecognized, then the resource is a code resource with a format of `guess`.) Because of this, there is no way to specify a resource type in the attribute list: it is always inferred from the format. Also, there's no default value for the format of a given resource type: the format is determined first, and it determines the type.

We will now consider each of the types of resources in more detail, as well as the various formats that they support. We will also discuss the supported attributes for each resource type. Resources have different default attributes based on their type, format and insertion method. These default attributes are important, especially for resources inserted as a span, since they cannot have an actual attribute list–only resources inserted as a figure can have an attribute list.

Images

The most common type of resource in most Markua documents will probably be image. Like any resource, an image can be inserted as a span or as a figure.

The following types of images are supported in Markua: GIF, PNG, JPEG, SVG and zipped SVG.

Extension	`format`	Description
`.gif`	`gif`	GIF image
`.png`	`png`	PNG image
`.jpg`	`jpeg`	JPEG image
`.jpeg`	`jpeg`	JPEG image
`.svg`	`svg`	SVG image
`.svgz`	`svgz`	Zipped SVG

⚠ TODO_LEANPUB - Support table captions (there should be one in the table above)

The syntax to insert an image is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for any type of image; inline resource locations are supported for SVG images only.

> Images in Markua are inserted in essentially the same way they are in Markdown for inline images. Again, the reference style image syntax of Markdown is **NOT** supported in Markua for images or any type of resource.

We will discuss the supported and the default attributes for images, and then show examples of images being inserted as spans and as figures for both local, web and inline images.

Supported Attributes for Images

`format`
: All resources support the format attribute. For images, the `format` is the specific image format. The legal set of formats is given in the `format` column in the above table.

`align`
: The `align` is the horizontal alignment. It can be `center` (the default), `left` or `right`.

`float`
: The `float` can be `none` (the default), `left`, `center` or `right`. The `float` trumps `align`. If `float` is

anything except its default of `none`, the value of `align` is ignored and the resource is positioned according to the value of the `float`, with the text flowing freely around it.

`position`
The `position` refers to vertical positioning on a page. The position only has an effect on PDF generation – it is ignored in EPUB and MOBI. It can be `near` (the default), `here`, `top` or `bottom`. The `near` position means to position the image near here, while the `here` position means to position the image exactly here. The `top` position means the top of the page; the `bottom` position means the bottom of the page. (If you're familiar with LaTeX, `near` is similar to `h` and `here` is similar to `H` or `h!`. Note that Markua deliberately does not specify anything about PDF output, to maximize implementation flexibility and encourage competition.)

`width` and `height`
The `width` and `height` can both be `auto` (the default), `fullbleed` or an integer between `1` and `100` inclusive. The default of `auto` means to respect the actual size of the image without overflowing into page margins. If only one of `width` and `height` are `auto`, the image aspect ratio is respected; if both `width` and `height` are specified, the image is resized accordingly. The value of `fullbleed` means to resize the image to full width or height of the page, ignoring margins. (In the HTML output this just adds a class of `fullbleed` and sets the width to `100%`, and CSS can be used to do whatever the author or book designer chooses to accomplish the fullbleed effect.) It is legal for either one or both of `width` and `height` to be `fullbleed`. The integer values between `1` and `100` are percentage widths of the content area of the page, which respects the margins.

`alt`
The `alt` is the alt text, to be displayed when the image cannot be shown. This can only be provided in square brackets for images inserted as spans and only in the attribute list for figures.

`caption`
The `caption` is the figure caption. It can only be present when the image is inserted as a figure.

Default Attributes for Images

The default attributes for an image inserted as a span are:

```
{align: left, float: none, position: here, width: auto, height: auto}
```

Since images inserted as a span cannot have their default attributes overridden, when an image is inserted as a span image, it must be positioned where it is inserted according to the `position: here` default attribute–it cannot be floated and repositioned by Markua processors.

The default attributes for an image inserted as a figure are:

```
{align: center, float: none, position: near, width: auto, height: auto}
```

These attributes can all be overridden via the attribute list on the figure.

Local Images

Span

```
Easy as `![pie](pie.png)`.
```

Figure

```
Here's a paragraph before the figure.

{alt: "a slice of chocolate cake"}
![A Piece of Cake](cake.jpg)

Here's a paragraph after the figure.
```

Web Images

Span

```
Easy as `![pie](http://markua.com/pie.png)`.
```

Figure

```
Here's a paragraph before the figure.

{alt: "a slice of chocolate cake"}
![A Piece of Cake](http://markua.com/cake.jpg)

Here's a paragraph after the figure.
```

Inline Images (SVG only)

Span

```
Red `<svg width="10" height="10"><rect x="0" y="0" width="10" height="10" fill="red"/></svg>`svg square.
```

Figure

~~~
Here's a paragraph before the figure.

{caption: "Earth From Space (Simplified)", alt: "a blue circle"}
```svg
<svg width="20" height="20">
  <circle cx="10" cy="10" r="9" fill="blue"/>
</svg>
```

Here's a paragraph after the figure.
~~~

Video

Unlike images, which are supported in most circumstances, with video files it's currently a bit of a crapshoot. There's currently a dominant proprietary format (H.264, or .mp4) and a new open source challenger (WebM). It's entirely likely that many ebook readers won't support either.

The following types of videos are supported in Markua: MP4 and WebM.

Extension	`format`	Description
`.mp4`	`mp4`	MP4 video
`.webm`	`webm`	WebM video

The syntax to insert a video is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for both video formats; inline resource locations for video are obviously not supported.

We will discuss the supported and the default attributes for videos, and then show examples of videos being inserted as spans and as figures for both local and web videos.

Supported Attributes for Video

`format`

All resources support the format attribute. For video, the `format` is the specific vidio format. The legal set of formats is given in the `format` column in the above table.

`poster`

The `poster` is the URL or path to an image which should be shown instead of the video before the video is played. If a Markua Processor is outputting some format where it is known that video resources are not supported, it must choose the poster to use as a replacement for the video. (Books are not just ebooks–books can also be printed on the fibers of trees that have been chopped down ("paper"), producing something called a "book". These "books", whether they are bound in a sturdy fashion ("hardcover books") or a flimsy fashion ("paperback books"), have one thing in common with respect to embedded video: they do not support it.

`alt`, `caption`, `align`, `float`, `position`, `width` and `height`

These attributes work exactly how they do for images. They apply to the image specified by the `poster` attribute, and to the size of the video once it starts playing. Once the video starts playing, it can obviously be made fullscreen by any player that supports it.

Since the `alt` is a fallback for if the `poster` image cannot be displayed, the video has two fallbacks: the `poster` image and then the `alt` text. (The use of video fallbacks, such as HTML5's `mediagroup` syntax or multiple `source` elements, is not supported in Markua.)

Default Attributes for Video

The default attributes for a video inserted as a span are:

```
{align: left, float: none, position: here, width: auto, height: auto}
```

The default attributes for a video inserted as a figure are:

```
{align: center, float: none, position: near, width: auto, height: auto}
```

These attributes can all be overridden via the attribute list on the figure.

Local Video

Span

```
Easy as `![pie](pie_eating_contest.mp4)`.
```

Figure

```
Here's a paragraph before the figure.

{alt: "a man eating an entire chocolate cake"}
![A Piece of Cake](cake_eating_contest.webm)

Here's a paragraph after the figure.
```

Web Video

Span

```
Introducing Markua: `![Peter's Markua talk](https://www.youtube.com/watch?t=105&v=VOCYL-FNbr0)`.
```

> Ironically, I'm writing this sentence almost a year after that talk was recorded, and I still haven't figured everything out about Markua.

Figure

```
Here's a paragraph before the figure.

{poster: http://img.youtube.com/vi/VOCYL-FNbr0/mqdefault.jpg}
!["Introducing Markua](https://www.youtube.com/watch?t=105&v=VOCYL-FNbr0)

Here's a paragraph after the figure.
```

Audio

Just like with video, the audio support in ebooks and on the web is more varied than for images. With audio, there are MP3, AAC, Ogg and WAV formats all in widespread use, and there are a number of other formats with supporters. It's entirely likely that many ebook readers won't support any of them.

The following types of audio resources are supported in Markua: MP3, AAC, WAV and Ogg Vorbis.

Extension	`format`	Description
`.mp3`	`mp3`	MP3 audio
`.m4a`	`aac`	AAC audio
`.aac`	`aac`	AAC audio
`.wav`	`wav`	WAV audio
`.wave`	`wav`	WAV audio
`.ogg`	`ogg`	Ogg Vorbis
`.oga`	`ogg`	Ogg Vorbis

Note that .mp4 is not supported as a file extension for MP4 AAC audio, since that is the file extension used for MP4 video.

The syntax to insert an audio file is the same compact and consistent syntax that is used for any resource. Local and web resource locations are supported for both audio formats; inline resource locations for audio are obviously not supported.

We will discuss the supported and the default attributes for audio files, and then show examples of audio being inserted as spans and as figures for both local and web audio files.

Supported Attributes for Audio

`format`
: All resources support the format attribute. For audio, the `format` is the specific audio format. The legal set of formats is given in the `format` column in the above table.

`poster`
: The `poster` is the URL or path to an image which should be shown instead of the audio. (Think of it like an "album" cover: many humans have never seen an LP; for them, an album cover is as much a metaphor as the floppy disk which represents a save button.) The poster can have a play button superimposed on it by the Markua Processor. A Markua Processor can also ignore the poster entirely and just show a generic player for the audio resource. If a Markua Processor is outputting some format where it is known that audio resources are not supported (e.g. "books"), it must choose the poster to use as a replacement for the audio.

`alt`, `caption`, `align`, `float`, `position`, `width` and `height`
: These attributes work exactly how they do for images. They apply to the image specified by the `poster` attribute, not to the audio itself.

Since the `alt` is a fallback for if the `poster` image cannot be displayed, the audio has two fallbacks: the `poster` image and then the `alt` text.

Default Attributes for Audio

The default attributes for audio are (with the exception of the type) the same as for video, and they all apply to the poster.

The default attributes for an audio inserted as a span are:

```
{align: left, float: none, position: here, width: auto, height: auto}
```

The default attributes for an audio inserted as a figure are:

```
{align: center, float: none, position: near, width: auto, height: auto}
```

These attributes can all be overridden via the attribute list on the figure.

Local Audio

Span

```
The talk audio highlights are here: `![](highlights.mp3)`.
```

Figure

```
The full version of the talk is here:

![](talk.m4a)
```

Web Audio

Span

```
The talk audio highlights are here: `![](http://markua.com/highlights.mp3)`.
```

Figure

```
The full version of the talk is here:

![](http://markua.com/talk.m4a)
```

Code

Code can be a local, web or inline resource, just like any other resource, and the same resource syntax applies to code as to all other resources.

> Markua's origins are in Leanpub Flavoured Markdown, which evolved based on Markdown but for computer programming books. So, everything you need to write a computer programming book is in Markua. In Leanpub Flavoured Markdown we had a special syntax for inserting code samples. It looked like `<<(code/some_code_file.rb)`. In Markua, however, code is just a resource, and that syntax is not supported.

As discussed, code cannot have alt text. It's just text. If any alt text is provided for a code resource it is ignored.

Markua specifies only one specific file extension, `.txt`, to be associated with the code type. However, Markua Processors must interpret all unspecified file extensions as specifying a resource of type code with a format of `guess`.

Extension	format	Description
.txt	text	Monospaced plain text
(other)	guess	Implementation-dependent

All code must be formatted as monospaced text by Markua Processors.

The `text` format means to not do any syntax highlighting as well.

The `guess` format is a request for the Markua Processor to guess at the programming language based on the file extension and/or the syntax of the code itself. Then, if the detected language corresponds to a particular programming language which the Markua Processor recognizes, and if the Markua Processor supports syntax highlighting, then it can format the resource as nicely syntax-highlighted code.

Syntax highlighting is optional in Markua Processors. If a Markua Processor does not support syntax highlighting, or if it cannot detect a matching supported programming language, then it must format the code as though the format was `text`–i.e. to format it as unformatted monospaced text.

Besides the `text` and `guess` values of the format attribute, you can also specify the programming language by setting the format attribute to a specific programming language. This is more reliable than `guess`. Unlike other resource types, Markua does not specify the complete set of the values of the `format` attribute–there are so many programming languages in the world, and new ones are added so frequently, that doing so would be impractical.

Supported Attributes for Code

`caption`
: All resources inserted as figures support the `caption` attribute.

`align`, `float`, `position`
: These attributes function how they do for images.

`format`
: For code, the `format` is the name of the programming language. There are two special types of `format` for code baked into Markua: `text` and `guess`.

`line-numbers`
: This determines whether the code sample shows line numbers. Legal values are `true` or `false`. The default value is `false`. Any value other than `true` is interpreted as `false`. Note that a code resource inserted as a span should be a single line of code, so this attribute does not make sense with spans.

`number-from`
: If line numbers are shown, this lets you override the starting number of the line numbers. The default value is `1`.

`crop-start`
: Sometimes it's desirable to only show part of a code resource defined in an external file as the code example. The `crop-start` and `crop-end` attributes let you accomplish this. The `crop-start` attribute defines the line which represents the first line included from the resource. For example, `{crop-start: 10, crop-end: 15, line-numbers: true, number-from: 10}` ensures that lines 10-15 are shown and are numbered as lines 10-15. The default value is 1, which is the first line of the file.

`crop-end`
: This attribute ends the range started with crop-start. The default value of crop-end is to be omitted, which is equivalent to specifying the last line of the file.

Default Attributes for Code

The default attributes for a code resource inserted as a span are:

```
{format: guess, align: left, float: none, position: here, line-numbers: off}
```

The default attributes for a code resource inserted as a figure are:

```
{format: guess, align: center, float: none, position: near, line-numbers: off}
```

The default attributes for a code resource inserted as a figure using tildes as the delimiter are:

```
{format: text, align: center, float: none, position: near, line-numbers: off}
```

Note that the difference between the default attribute lists with the backtick-delimited and tilde-delimited inline resources is just a difference of the default format (`guess` vs. `text`). The `guess` format works better for code; the `text` format works better for poetry and for code that you don't want the language guessed at. You can specify any attributes you wish with either delimiter, and specified attributes override default ones. The only reason there are different defaults are as syntactic sugar.

Local Code Resources

Span

```
Hello in Ruby is a simple ![](hello.rb) statement.
```

Note that in this example there can be no text inside the square brackets, since in a span the square brackets are for alt text, and with a code resource alt text is not supported.

Figure

```
This will be a type of code and a format of `guess` since the file extension is not specified:

![Hello World in Ruby](hello.rb)

That is equivalent to:

{format: guess}
![Hello World in Ruby](hello.rb)

If you don't want to take chances you can do this:

{format: ruby}
![Hello World in Ruby](hello.rb)

Here's a paragraph at the end.
```

Note that in the above examples of figures, the text in the square brackets is the figure caption, like it is in all figures.

Web Code Resources

This is identical to how local code resources work, including the significance of file extensions. The only differences is that the files are on the web.

Span

```
Hello in Ruby is a simple ![](http://markua.com/hello.rb) statement.
```

Note that in this example there can be no text inside the square brackets, since in a span the square brackets are for alt text, and with a code resource alt text is not supported.

Figure

```
This will be a type of code and a format of `guess` since the file extension is not specified:

![Hello World in Ruby](http://markua.com/hello.rb)

That is equivalent to:

{format: guess}
![Hello World in Ruby](http://markua.com/hello.rb)

If you don't want to take chances you can do this:

{format: ruby}
![Hello World in Ruby](http://markua.com/hello.rb)

Here's a paragraph at the end.
```

Note that in the above examples of figures, the text in the square brackets is the figure caption, like it is in all figures.

Inline Code Resources

Inline code resources are the most flexible and appropriate way to insert code as a span resource, and the most straightforward way to add short code examples as figures.

Span

The format of an inline code resource inserted as a span is:

````
Yada yada `some code here`optional_format yada yada.
````

The optional format specifies the programming language used. If it is not provided, the format defaults to `guess`, as shown in the following examples:

````
Hello in Ruby is a `puts "hello"` statement.

Hello in Ruby is a `puts "hello"`guess statement.

Hello in Ruby is a `puts "hello"`ruby statement.

Hello in Ruby is a `puts "hello"`text statement.
````

The first and second statement above are functionally identical since the default format of an inline resource with no format specified is `guess`. The third statement forces the format to be `ruby`. The fourth statement forces it to be plain monospaced code with no syntax highlighting.

> The syntax of adding inline code resources as spans is identical to the code span syntax of Markdown.

Figure

~~~
This will be a type of code and a format of `guess` since the format is not specified:

```
puts "hello"
```

That is equivalent to:

```guess
puts "hello"
```

If you don't want to take chances you can do this:

```ruby
puts "hello"
~~~

~~~
```

If you don't like syntactic sugar you can do:

{format: ruby}
```
puts "hello"
```
~~~

If you want a figure caption, you can add it to the attribute list with any of the above. For example:

~~~
Some paragraph.

{caption: "Hello World in Ruby"}
```ruby
puts "hello"
```

Some paragraph.
~~~

The syntax for inline code resource figures is similar to the "fenced code blocks" syntax of many Markdown extensions, such as PHP Markdown Extra and GitHub Flavoured Markdown.

The "four space indent" method of creating code blocks in Markdown is not supported in Markua. You cannot create an inline code figure this way. It may be slightly more pleasant to read, but it's obnoxious as hell to write, and it has bad effects on the rest of the specification.

Typewriter Poetry with Code Resources

If you want to write "typewriter poetry" in the style of e e cummings, in which all the indentation and other whitespace matters, you can accomplish this with a code resource whose format is `text`.

There are three ways to insert such a code resource inline:

1. by specifying the `text` format in metadata
2. by specifying the `text` format after the three backticks or three tildes
3. by using three tildes to insert the inline code resource

Examples:

Using three tildes:

````
~~~
i
  am
      a        cat
            a        l a z y          cat

me   ow
~~~
````

Using three backticks plus the `text` format via syntactic sugar:

~~~
```text
i
  am
      a        cat
            a        l a z y          cat

me   ow
```
~~~

Using three backticks plus the `text` format specified in an attribute list:

~~~
{format: text}
```
i
  am
      a        cat
            a        l a z y          cat

me   ow
```
~~~

This use case is why the default format for three backticks is `guess` and the default format for three tildes is `text`: you wouldn't want your poem inadvertently being syntax highlighted like code, and the three backticks plus `text` version does not look very poetic!

(Determining the artistic merit of this "poem" is left as an exercise for the reader.)

Line Wrapping in Code Resources

Code resources should have newlines added by the author to ensure that automatic line wrapping is not relied upon. Markua Processors may wrap lines to ensure that all code is visible on a page, and *may* add continuation characters (like the backslash \ character) in the output to indicate that a line has been automatically wrapped. However, adding a continuation character is not a requirement.

Math

Math can be a local, web or inline resource, just like any other resource, and the same resource syntax applies to code as to all other resources.

> Just as with code, in Leanpub Flavoured Markdown we had a special syntax for inserting math. It looked like `{$$}...{/$$}`, which looks nice to people who like LaTeX, and looks like nothing else in Markdown. In Markua, however, math is just another resource, and that LaTeX-inspired syntax for wrapping math resources is not supported.

Math resources can be either LaTeX math or MathML.

Extension	`format`	Description
`.tex`	`latex`	LaTeX math
`.mathml`	`mathml`	MathML math

Note that Markua procesors do not need to support either Math markup language fully in order to be compliant: if they choose to, they can simply render the text as a code resource of format `text`.

Supported Attributes for Math

`caption`
: All resources inserted as figures support the `caption` attribute.

`align`, **`float`**, **`position`**
: These attributes function how they do for images. (In fact, a Markua Processor may choose to transform the math into an image, for maximum ebook reader compatibility.)

`format`
: For math, the `format` is the name of the syntax used to write the mathematical equations. There are two special types of `format` for math baked into Markua: `tex` for LaTeX math and `mathml` for MathML math.

`alt`
: The `alt` is the alt text, to be displayed when the mathematical equations cannot be shown. This can only be provided in square brackets for mathematical equations inserted as spans and only in the attribute list for figures.

Default Attributes for Math

The default attributes for a math resource inserted as a span are:

```
{align: left, float: none, position: here, width: auto, height: auto}
```

Since a math resource inserted as a span cannot have its default attributes overridden, when a math resource is inserted as a span, it must be positioned where it is inserted according to the `position: here` default attribute–it cannot be floated and repositioned by Markua processors.

The default attributes for a math resource inserted as a figure are:

```
{align: center, float: none, position: near, width: auto, height: auto}
```

These attributes can all be overridden via the attribute list on the figure.

Local Math Resources

Span

```
This is ![too large to fit in the alt text](fermat_proof.tex) my proof.
```

Figure

```
Here's a paragraph before the figure.

{alt: "too large to fit in the alt text"}
![Proof of Fermat's Last Theorem](fermat_proof.webm)

Here's a paragraph after the figure.
```

Web Math Resources

This is identical to how local math resources work, including the significance of file extensions. The only differences is that the files are on the web.

Span

```
This is ![too large to fit in the alt text](http://markua.com/fermat_proof.webm) my proof.
```

Figure

```
Here's a paragraph before the figure.

{alt: "too large to fit in the alt text"}
![Proof of Fermat's Last Theorem](http://markua.com/fermat_proof.tex)

Here's a paragraph after the figure.
```

Note that in the above examples of figures, the text in the square brackets is the figure caption, like it is in all figures.

Inline Math Resources

Inline math resources are the most flexible and appropriate way to insert math as a span resource, and the most straightforward way to add short math examples as figures. Both LaTeX and MathML can be inlined either as spans or figures, but only LaTeX is terse enough for it to be practical.

Span

The format of an inline math resource inserted as a span is:

````
Yada yada `some math here`optional_format yada yada.
````

The optional format specifies the math format used. While it's technically optional, it's not optional if you want math: you either need to say `latex` or `mathml`. Otherwise, if the format is not provided, the format defaults to `guess` and the resource is interpreted as code, not math.

````
One of the kinematic equations is `d = v_i t + \frac{1}{2} a t^2`latex this one.
````

Since these are span elements, LaTeX math will look a lot better than MathML, since it's a lot more terse. However, both are supported.

Figure

LaTeX math:

~~~
Here's a paragraph before the figure.

```latex
\left|\sum_{i=1}^n a_ib_i\right|
\le
\left(\sum_{i=1}^n a_i^2\right)^{1/2}
\left(\sum_{i=1}^n b_i^2\right)^{1/2}
```

Here's a paragraph after the figure.
~~~

MathML math:

~~~
Here's an example from [Wikipedia](http://en.wikipedia.org/wiki/MathML):

{type: math, format: mathml}
```
<apply>
  <plus/>
  <apply>
    <times/>
    <ci>a</ci>
    <apply>
      <power/>
      <ci>x</ci>
      <cn>2</cn>
    </apply>
  </apply>
  <apply>
    <times/>
    <ci>b</ci>
    <ci>x</ci>
  </apply>
  <ci>c</ci>
</apply>
```

Here's a paragraph after the figure.
~~~

Lists

Like Markdown, Markua supports ordered (i.e. numbered) and unordered (i.e. bulleted) lists.

Unlike Markdown, Markua also supports definition lists. Definition lists are discussed in the next section–although they are called "lists", their formatting and function is very different than unordered and ordered lists.

If you are familiar with Markdown, as you read the sections below you will note a number of differences between Markua lists and Markdown lists.

Unordered Lists (i.e. Bulleted Lists)

Markua lets you make an unordered list by starting each list item with either an asterisk (*) or a hyphen (-), followed by 1-4 spaces or 1 tab, followed by text content. You can't mix and match asterisks and hyphens in the same list. Here's how this looks:

You can build a list out of items starting with an asterisk and one space:

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

You can build a list out of items starting with a hyphen and one tab:

```
-    one
-    two
-    three
```

(My editor converts tabs to spaces, so that's actually 4 spaces, which also works.)

> Markdown also lets you use a plus sign (+) before each item, but Markua does not. Having three syntaxes is just overkill, so Markua drops the plus–it's far less common than the asterisk and hyphen. This is the same decision made by GitHub Flavored Markdown, which also supports[a] * and - but not +. Markua could have chosen only the asterisk (*) or the hyphen (-) and only one amount of spaces or tabs after it, but this would be too prescriptive. Markua is as opinionated as possible, but that would have been overkill.
>
> [a]https://help.github.com/articles/markdown-basics/

These are the specific rules for unordered lists:

- Start each list item a bullet which is either an asterisk (*) or a hyphen (-).
- You can't mix and match asterisks and hyphens in the same list.
- Between 1 and 4 spaces or one tab is allowed after each bullet.
- You cannot mix tabs and spaces: if a tab is used after a bullet, tabs must be used after each bullet.

- If spaces are used after each bullet, the same number of spaces must be used after each bullet in the list.
- A varied number of spaces or tabs or a mixture of tabs and spaces does not create an unordered list. Instead, it creates a paragraph with line breaks inside it, which produces break tags in HTML.
- A single element unordered list is a list: although it is a pretty stupid list, treating it as a literal paragraph starting with an asterisk, plus or hyphen would be even stupider.

In terms of style guidance, the preferred bullet type is the asterisk, and the preferred whitespace after the bullet is one space.

Ordered Lists (i.e. Numbered Lists)

Markua defines many more types of ordered list numbering than Markdown.

> Despite the fact that all lists actually have an order (otherwise they would not be lists!), Markua is sticking with the HTML terminology for “ordered” vs. “unordered” lists, instead of “numbered” vs. “bulleted” lists. Even though these feel more like programming terms, HTML terminology is so widely understood that using any other terminology is distracting. (Also, since ordered lists support numbering by letters, the term “ordered” is superior to “numbered” in some senses.)

In Markdown, the only type of numbering supported is decimal numbering starting from 1. If you need any more features, you need to use inline HTML. However, since Markua does not support inline HTML, Markua provides more list features.

In Markua, an ordered list can vary the following:

1. Numbering system
2. Numbering direction (ascending or descending)
3. Initial number (or letter, or Roman numeral)

The following choices of numbering system are supported:

1. Decimal numbers
2. Uppercase letters
3. Lowercase letters
4. Uppercase Roman numerals
5. Lowercase Roman numerals

Unlike in Markdown, in Markua **the number that begins the list in the manuscript is the number that begins the list in the output**.

To make an ordered list in Markua, you start one or more consecutive lines with either a consecutive number or the same number (or letter or roman numeral), followed by between 1 and 4 spaces or 1 tab, followed by text content.

Since Markua supports letters and Roman numerals as well as decimal numbers to start lists, the rules about using consecutive numbers or the same number are actually a bit complex.

Note that there are one or more spaces after the period. Unlike with an unordered list, the number of spaces does not need to be the same after each item (since numbers may have 2 or more digits, so a variable number of spaces may be desired to line everything up).

> I had really wanted to support parentheses after the number as well as periods, since to me `a)` looks a lot better than `a.` does. However, there's no *good* way to do this in HTML, so I'm giving up and not doing it. If you make a lettered list with parentheses, it just turns into a paragraph with a bunch of newlines for the list items, so you basically get what you want, just not using HTML constructs.

This set of examples shows many of the normal use cases of lists with the various numbering systems. For the edge cases, see the next sections. For the mapping to HTML lists, see the next chapter.

Ascending decimal numbers starting from 1:

```
1. foo
2. bar
3. baz
```

Ascending decimal numbers starting from a higher number:

```
9.  foo
10. bar
11. baz
```

Descending decimal numbers:

```
3. foo
2. bar
1. baz
```

Identical decimal numbers for the lazy (producing 1, 2, 3):

```
1. foo
1. bar
1. baz
```

Ascending lowercase letters:

```
a. foo
b. bar
c. baz
```

Ascending uppercase letters:

```
I. foo
J. bar
K. baz
```

Ascending uppercase Roman numerals:

```
I.   foo
II.  bar
III. baz
```

In Markua, A Single Element Ordered List is Not a List

Markdown has the interesting combination of supporting one element lists and ignoring the number that a list starts with. This means it's possible to inadvertently start a numbered list by beginning any line with a number followed by a period. The example that John Gruber cites[13] is the following:

```
1986. What a great season.
```

This would produce a single element numbered list starting with `1.` – which in my opinion, is a blatant violation of the Principle of Least Surprise.

Now, in Markua, if we supported single element ordered lists, this would produce a single element numbered list starting with `1968.` – which would be almost as bad as in Markdown.

(By the way, there is a very gross workaround in Markdown: you prefix the period with a backslash. So, you'd write `1968\. What a great season.` to avoid this.)

But what to do?

The answer is simple: in Markua, **single element ordered lists are not lists**. This is true for ordered lists only – single element unordered lists and single element definition lists are both lists. (In the case of an unordered list it's stupid, but the alternative is stupider. In the case of a definition list there's actually a legitimate usage of a single element definition list: a single definition.)

To me, the notion of a *list* is something with more than one thing in it. If I have a list of chores to do, there are many of them. If I have one chore to do, I don't call it a list of chores–I call it a chore.

So, in Markua, the automatic creation of an ordered list only happens if you have two or more lines starting with numbers and periods.

This still has some possibly incorrect interpretations, but these will be a lot more rare. This matters: if you get burned by the automatic list creation, and you feel that you have to think about whether you can start a sentence with a number, then writing in Markua will feel more like programming than writing.

[13]http://daringfireball.net/projects/markdown/syntax

This is a microcosm of the difference between Markua and Markdown: since Markdown is a way of producing HTML, it is biased toward producing HTML constructs wherever it is most straightforward to do so. Markdown is a way of writing books and documents, and it only produces HTML constructs like lists when it is almost certainly appropriate to do so. So, whereas Markdown tries to make HTML wherever possible, Markua respects what the author wrote, and only substitutes HTML constructs where doing so is lossless in the text (period and parentheses are not equal) and unambiguous (a single element is probably not a 1 element list).

Ordered List Numbering Rules

If you are an author who is just trying to write straightforward Markua, you can probably skip this section.

There are a number of rules about what does and does not trigger the creation of an ordered list. These rules are designed to help authors stay sane when writing in Markua, and also to help implementors of Markua processors stay sane. In all cases, the first number, letter or Roman numeral is the start of the list numbering. This never changes.

If, because of the correct application of these rules, an ordered list is not created, what happens is that a paragraph is created instead, with a break tag for each single newline. In this instance, the output directly matches the input, including all the numbering.

In theory, Markua supports creating an ordered list when either of the following two conditions occur:

1. Every item in the list begins with a consecutive number, either increasing or decreasing.
2. Every item in the list begins with the same number.

This is not actually quite true.

See, the notion of "consecutive" is easy to explain and to check for decimal numbers, either in increasing or decreasing order. `1, 2, 3, ...`, `456, 457, 458, ...`, or `5, 4, 3` are all easily verified as consecutive, both by humans and computers.

However, Markua also supports ordered lists using uppercase and lowercase alphabetical numbering, as well as uppercase and lowercase Roman numeral numbering, and in increasing or decreasing order.

In the case of alphabetical numbering, checking "consecutive" is harder. It's relatively straightforward for the first 26 items, but then all bets are off. In case you're curious, here is the sequence used in HTML for increasing alphabetical numbering:

`a, b, ..., z, aa, ab, ..., az, ba, bb, ... bz, za, zb, ..., zz, aaa, aab, ..., aaz, aba, abb, ...`

In terms of Roman numerals, it's even worse. I'm sure that someone in a Classics department (or in the NFL) knows the algorithm by heart, but I don't.

So, what are Markua authors and implementors of Markua processors to do?

Markua makes this simpler by specifying the following rules.

Markua *actually* supports creating an ordered list based on a different set of conditions based on the numbering type.

For decimal numbers, Markua actually supports creating an ordered list when either of the following two conditions occur:

1. Every item in the list begins with a consecutive number, either increasing or decreasing.
2. Every item in the list begins with the same number.

For uppercase and lowercase alphabetical numbering, Markua actually supports creating an ordered list when either of the following two conditions occur:

1. Every item in the list begins with a consecutive letter, followed by a period, and the lettering does not extend past `z.` for lowercase or `Z.` for uppercase. (This way, implementors of Markua processors don't need to check the `aa`, case, and authors writing in Markua don't need to remember it.)
2. Every item in the list begins with the same single letter, followed by a period.

The term "single letter" means that consecutive `c.`s start an ordered list which starts with `c, d, e, ...`, but that consecutive `aa.`s or `aba.`s do not start an ordered list. Again, this greatly simplifies life (you're welcome!) for implementors of Markua processors, who otherwise would need to figure out what number in the sequence `aba` is, in order to generate the correct HTML.

For uppercase and lowercase Roman numeral numbering, Markua actually supports creating an ordered list when either of the following two conditions occur:

1. Every item in the list begins with a consecutive Roman numeral, followed by a period, and the lettering does not extend past `xii.` for lowercase Roman numerals or `XII.` for uppercase Roman numerals. To be clear, this means that the only valid consecutive lowercase Roman numerals are `i, ii, iii, iv, v, vi, vii, viii, ix, x, xi, xii`, and that the only valid consecutive uppercase Roman numerals are `I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII`.
2. Every item in the list begins with `i.` for lowercase Roman numerals or `I.` for uppercase Roman numerals.

The combination of the rules for Roman numeral numbering means that you can have an arbitrarily long Roman numeral list starting from `i.` or `I.`, but that implementors of Markua processors don't need to write code to figure out whether `MCMLXXV` is a valid Roman numeral, and what the consecutive Roman numeral sequence after it is.

If you're wondering why I picked `xii` and `XII` for the last consecutive lowercase and uppercase Roman numerals to respect: this is Roman numeral 12, and the main usage of Roman numerals in modern life is in clock faces.

Speaking of clock faces, I have bad news for fans of watches and antique clocks:

- The only supported version of the Roman numeral `4` is the subtractive `iv` (in lowercase) or `IV` (in uppercase); the additive `iiii` or `IIII` form is not supported.
- The only supported version of the Roman numeral `9` is the subtractive `ix` (in lowercase) or `IX` (in uppercase); the additive `viiii` or `VIIII` form is not supported.

It turns out that the question of additive versus subtractive forms of Roman numerals is actually interesting; see this article[14] and this Wikipedia entry[15] for a starting point.

In Markua, list numbers must **all** either be **consecutive**, given the type of numbering that is used, or **the same as the first number**. Otherwise, the list is interpreted as a paragraph with a bunch of break tags in it, and lines starting with the numbers given. The principle is the following: the numbers which are shown in the manuscript must be the same as those in the output, or the numbers must be clearly intended to be numbered list numbers. Otherwise, a list will not be produced.

If you want to prevent a list from being created with consecutive numbered items separated by single newlines, the backslash escape in front of the period used in Markdown to prevent lists also works in Markua. However, it is needed a lot less often.

Multi-Paragraph List Items

If lists were simple, they would contain be a bunch of simple one-line list items like:

```
* This is simple.
* So is this.
* This too!
```

However, in Markdown, list items can contain multiple paragraphs, and it makes sense for Markua to support this. If a list is going to contain list items with multiple paragraphs, then there should be blank lines between each list item.

As with Markdown, every paragraph in a list item after the first one must be indented by four spaces or one tab. This lets the Markua Processor know that the list is not over, but is instead being continued.

The following is legal Markdown and legal Markua:

```
Here's a paragraph before the list.

1.  This is the first paragraph in the first list item. Yay!

    This is a second paragraph in the first list item.

2.  The second list item is simple.

3.  The third list item has three paragraphs.
    This is still part of the first paragraph.

    Here's the second paragraph in the third list item.
```

[14]http://mentalfloss.com/article/24578/why-do-some-clocks-use-roman-numeral-iiii
[15]https://en.wikipedia.org/wiki/Roman_numerals

```
    Here's the third paragraph in the third list item.

Here's a paragraph after the list.
```

Just like with normal non-list-item text, a single newline functions as a forced line break within the list item in question. However, to encourage the text to be lined up nicely in the list item, any leading whitespace after the linebreak is ignored. This is different from the whitespace being preserved after single newlines in paragraphs. That rule exists to support poetry, and if there's one thing that lists are not, it's poetry.

Nested Lists

Lists can nest lists.

⚠ TODO_LEANPUB - support nested lists

To do this, simply insert the list nested inside a list item by 2 spaces for each level of nesting.

There is no limit to the levels of nesting, and you can nest any kind of list inside any kind of list.

```
* Foo
  1. One
  2. Two
    a. This is 2a
    b. This is 2b
  3. Three
* Bar
  i.   Lorem
  ii.  Ipsum
  iii. Dolor
* Baz
```

Nesting Block Elements in List Items

Nested lists are just an example of nesting block elements inside list items.

Other block elements such as figures can also be inserted nested inside list items. This is true for both single paragraph list items (with no blank lines between them) and multiple-paragraph list items (with a single blank line between each list item).

⚠ TODO_LEANPUB - support this

Single paragraph list items

~~~
* Foo
  ```ruby
  puts "hello"
  ```
* Bar
  {alt: "too large to fit in the alt text"}
  ![Proof of Fermat's Last Theorem](fermat_proof.webm)
* Baz
~~~

In the singlem paragraph list items case, indenting the block element is optional but recommended.

Multiple paragraph list items

~~~
* This is paragraph 1 in the first list item.

  ```ruby
  puts "hello"
  ```

  This is paragraph 2 in the first list item.

* This is paragraph 1 in the second list item.

  {alt: "too large to fit in the alt text"}
  ![Proof of Fermat's Last Theorem](fermat_proof.webm)

  This is paragraph 2 in the second list item.

* Baz
~~~

In the multiple paragraph list items case, you need to indent the block element by the same four spaces or one tab that you indent the subsequent paragraphs in the list item by, in order to indicate the continuation of the list.

Definition Lists

Definition lists are supported in Markua. Although some people[16] don't see the value in definition lists in HTML, I strongly believe in their value. Specifically, with the rise of mobile and the narrower screen reading experience becoming the new default, I see definition lists as being more important than tables. But instead of just being two columns, the idea of a definition list has actual meaning.

To define a definition list in Markua, use the following syntax:

[16]http://meta.stackexchange.com/questions/72395/is-it-possible-to-have-definition-lists-in-markdown

```
term 1
: definition 1

term 2
: definition 2
```

A definition list can define multiple definitions for a term:

```
term 1
: definition 1a
: definition 1b

term 2
: definition 2
```

Note that a single term definition list is a definition list, regardless of how many definitions for the term exist.

```
term
: definition
```

Inserting Block Elements Inside Paragraphs

In Markdown, lists and other block elements cannot be nested inside a paragraph. Instead, they all function like top-level siblings to a paragraph. This makes sense since Markdown is a way of producing HTML, and in HTML5 the W3C has told anyone who wants to embed a list inside a paragraph to go pound sand[17]:

> "The solution is to realise that a paragraph, in HTML terms, is not a logical concept, but a structural one. In the fantastic example above, there are actually five paragraphs as defined by this specification: one before the list, one for each bullet, and one after the list."

So, basically, you can't embed a `<ul>` or `<ol>` inside a `<p>` tag in HTML5. Since Markdown is a way of producing HTML, you can't do it in Markdown either.

In Markdown, a block element like a list or figure is a sibling of the paragraphs before or after it:

```
This is paragraph one.

* this is a
* sibling of
* paragraphs one and two

This is paragraph two.
```

[17]http://www.w3.org/TR/html5/grouping-content.html#the-p-element

Markua supports this method of inserting block elements.

However, Markua is more than just a way of producing HTML. Markua is a way of authoring books and documents. HTML is just one of the output formats of Markua. And in books, there is a long history of embedding lists, code samples and figures in the middle of a paragraph–not in a separate top-level element, in the actual middle of the paragraph. So, Markua supports this too.

So, many block elements such as lists and figures can also be inserted nested in a paragraph. The way this is done is to add single newlines instead of blank lines.

This example shows a list nested in the middle of a paragraph:

````
This is paragraph one.

This sentence is in paragraph two.
* this list is
* part of
* paragraph two
This sentence is also in paragraph two.

This is paragraph three.
````

This example shows a figure which is an inline code resource nested in the middle of a paragraph:

~~~
This is paragraph one.

This sentence is in paragraph two.
```ruby
puts "hello world"
```
This sentence is also in paragraph two.

This is paragraph three.
~~~

This example shows a figure which is a local code resource nested in the middle of a paragraph:

````
This is paragraph one.

This sentence is in paragraph two.
{format: ruby}
![Hello, World](hello.rb)
This sentence is also in paragraph two.

This is paragraph three.
````

This example shows figures which are local or web image resources nested in the middle of paragraphs. Note that like with all figures, the square bracket text in both cases is the figure caption (not alt text):

```
This is paragraph one.

This sentence is in paragraph two.
![A Piece of Cake](cake.jpg)
This sentence is also in paragraph two.

This sentence is in paragraph three.
{alt: "a slice of chocolate cake"}
![A Piece of Cake](http://markua.com/cake.jpg)
This sentence is also in paragraph three.

This is paragraph four.
```

It is possible to insert a block element as the last part of a paragraph by adding a single newline before it, but a blank line after it:

```
This is paragraph one.

This sentence is in paragraph two.
* this list is
* part of
* paragraph two

This is paragraph three.
```

This works for most block elements, like lists, figures and tables. It does **NOT** work for definition lists, since those rely on blank lines in between elements.

Tables

Markdown does not specify a table syntax. Since Markdown supports inline HTML, it does not need to–if you want a table, you can simply use an inline HTML table.

Markua does not support inline HTML, so it needs a table syntax. Actually, Markua has *two* table syntaxes: simple tables and complex tables.

Both of them are somewhat annoying to type. However, it's not actually so bad–unless you make columns that are too narrow at first, and then you need to widen them. Also, there will probably be tooling that evolves to help with tables–especially in the more complex cases.

Simple Tables

PHP Markdown Extra[18] defines a pretty good table syntax, which is supported and extended by GitHub Flavored Markdown[19]. So Markua supports this syntax since it is widely used. These tables are called "simple tables" in Markua.

At its most basic, a simple table looks like this:

[18]https://michelf.ca/projects/php-markdown/extra/#table
[19]https://help.github.com/articles/github-flavored-markdown/

```
Header A   | Header B   | Header C
-----------|------------|-----------
Content A1 | Content B1 | Content C1
Content A2 | Content B2 | Content C2
Content A3 | Content B3 | Content C3
```

You can also start and end the columns with pipes:

```
| Header A   | Header B   | Header C   |
|------------|------------|------------|
| Content A1 | Content B1 | Content C1 |
| Content A2 | Content B2 | Content C2 |
| Content A3 | Content B3 | Content C3 |
```

Finally, you can specify left, center and right alignment of columns with colons in the header separator row. The following columns are left-, center-, and right-aligned respectively:

```
| Header A    | Header B     | Header C    |
|:------------|:------------:|------------:|
| Content A1  | Content B1   | Content C1  |
| Content A2  | Content B2   | Content C2  |
| Content A3  | Content B3   | Content C3  |
```

Simple tables work fine in most cases, and they are the simplest to type.

Complex Tables

Complex tables are slightly more effort to type than simple tables. However, they're just as easy to read, and they have more features. These include being able to individually align cells and to specify a cell that spans multiple rows or multiple columns.

At its most basic, a complex table looks like this:

```
|============|============|============|
| Header A   | Header B   | Header C   |
|============|============|============|
| Content A1 | Content B1 | Content C1 |
|------------|------------|------------|
| Content A2 | Content B2 | Content C2 |
|------------|------------|------------|
| Content A3 | Content B3 | Content C3 |
|============|============|============|
```

The header is separated from the body cells by a row of = signs and `|` characters. Every row of cells is separated by a row of `-` signs and `|` characters. The `|` characters **MUST** line up and **MUST** start and end each line.

> My recommended approach is to write the table first, and then copy and paste the row separators. If you get the column widths wrong, updating row separators is a headache, and you'll end up copying and pasting them anyway.

You can also specify left, center and right alignment of columns with colons in the header separator row, just as with simple tables. The following columns are left-, center-, and right-aligned respectively:

```
|=============|==============|=============|
| Header A    | Header B     | Header C    |
|:============|:============:|============:|
| Content A1  | Content B1   | Content C1  |
|-------------|--------------|-------------|
| Content A2  | Content B2   | Content C2  |
|-------------|--------------|-------------|
| Content A3  | Content B3   | Content C3  |
|=============|==============|=============|
```

You can have more than one row of content in cells, and there is no ambiguity since each row of cells is separated with a row of `-` signs and `|` characters:

```
|=============|==============|=============|
| Header A    | Header B     | Header C    |
|:============|:============:|============:|
| Content A1  | Content B1   | Content C1  |
| is really   | is longer    |             |
| really      |              |             |
| long        |              |             |
|-------------|--------------|-------------|
| Content A2  | Content B2   | Content C2  |
|-------------|--------------|-------------|
| Content A3  | Content B3   | Content C3  |
|=============|==============|=============|
```

You can specify colspan just by skipping a `|` in the appropriate place:

```
|=============|==============|=============|
| Header A    | Header B     | Header C    |
|:============|:============:|============:|
| Content A1 and B1 Merged   | Content C1  |
|-------------|--------------|-------------|
| Content A2  | Content B2   | Content C2  |
|-------------|--------------|-------------|
| Content A3  | Content B3   | Content C3  |
|=============|==============|=============|
```

You can specify rowspan just by skipping a part-row of `-` signs in the appropriate place:

```
|=============|==============|=============|
| Header A    | Header B     | Header C    |
|:============|:============:|============:|
| Content A1  | Content B1   | Content C1  |
| and A2      |--------------|-------------|
| merged      | Content B2   | Content C2  |
|-------------|--------------|-------------|
| Content A3  | Content B3   | Content C3  |
|=============|==============|=============|
```

You can also individually align cells with colons in the `-` part-row **above** them. This alignment overrides any alignment already on the column, and it works for merged cells too:

```
|=============|===============|=============|
| Header A    | Header B      | Header C    |
|:============|:=============:|============:|
| Content A1 and B1 Merged    | Content C1  |
|------------:|:--------------|-------------|
| Content A2  | Content B2    | Content C2  |
|:-----------:| and B3 merged |-------------|
| Content A3  |               | Content C3  |
|=============|===============|=============|
```

In that table, column A is left-aligned, column B is center-aligned and column C is right-aligned. However, cell A2 is right-aligned, cell A3 is center-aligned and merged cell B2 and B3 is left-aligned.

Tables can have Captions and Other Attributes

Regardless of whether they are simple or complex, tables can have captions and other attributes which are specified in an attribute list. The syntax for the attribute list is identical to that used for figures–but tables are not figures, they are tables.

This simple table has a caption:

```
{caption: "A Simple Table"}
| Header A   | Header B   | Header C   |
|------------|------------|------------|
| Content A1 | Content B1 | Content C1 |
| Content A2 | Content B2 | Content C2 |
| Content A3 | Content B3 | Content C3 |
```

This complex table has a caption as well as align, width and top attributes:

```
{caption: "A Complex Table", align: left, width: full, position: top}
|============|==============|============|
| Header A   | Header B     | Header C   |
|:===========|:============:|===========:|
| Content A1 and B1 Merged  | Content C1 |
|-----------:|:-------------|------------|
| Content A2 | Content B2   | Content C2 |
|:----------:| and B3 merged |------------|
| Content A3 |              | Content C3 |
|============|==============|============|
```

Supported Attributes for Tables

`align`
The `align` is the horizontal alignment. It can be `center` (the default), `left` or `right`.

`position`
The `position` refers to vertical positioning on a page. The position only has an effect on PDF generation – it is ignored in EPUB and MOBI. It can be `near` (the default), `here`, `top` or `bottom`. The `near` position means to position the table near here, while the `here` position means to position the table exactly here. The `top` position means the top of the page; the `bottom` position means the bottom of the page. (If you're familiar with LaTeX, `near` is similar to `h` and `here` is similar to `H` or `h!`. Note that Markua deliberately does not specify anything about PDF output, to maximize implementation flexibility and encourage competition.)

`width`
The `width` can be either `auto` (the default) or `full`. The `auto` width means to size the table based on its columns; the `full` width means to make it 100% of the content area of the page, respecting margins.

`caption`
The `caption` is the table caption.

Block Elements

We've already seen many examples of block elements in this specification, including paragraphs, headings and figures. These are the rest of them…

Blockquotes (`> `)

Blockquotes in Markdown are created by prefacing lines with `> `, i.e. a greater than character followed by a space:

```
This is a paragraph.

> This is a blockquote
> which is outside the paragraph.

This is a paragraph.
```

Blockquotes in Markua are created in one of two ways:

1. By prefacing lines with `> `, i.e. a greater than character followed by a space.
2. By wrapping the blockquote in `{blockquote}` … `{/blockquote}`

Option #1 is preferable for short quotes; option #2 is easier on authors for really long quotes.

Like figures and tables, blockquotes can be inserted in the middle of a paragraph or as a sibling of it.

These Markua blockquotes are siblings of the paragraphs:

```
This is a paragraph.

> This is a blockquote
> which is outside the paragraph.

This is a paragraph.

{blockquote}
This is a blockquote
which is outside the paragraph.
{/blockquote}

This is a paragraph.
```

These Markua blockquotes are nested in the paragraph:

```
This is a paragraph.

This is a second paragraph.
> This is a blockquote
> which is inside the second paragraph.
This is part of the second paragraph.
{blockquote}
This is a blockquote
which is inside the second paragraph.
{/blockquote}
This is part of the second paragraph.

This is a paragraph.
```

A blockquote can contain other block-level elements, most commonly paragraphs.

If you are using the `{blockquote}` … `{/blockquote}` approach, this is trivial: just pretend you're in a normal paragraph, and the syntax is the same.

If you are using the Markdown approach of `> `, then to start a new block level element within a blockquote, just put a line starting with a `>` followed by a optional space, followed by the block level element. It is equivalent to placing a `> ` in front of every line of the paragraphs.

blockquotes can be multi-paragraph. To create a multi-paragraph blockquote, you need to separate each paragraph with a line containing a `>` and whitespace only.

Single newlines inside a blockquote do not start new paragraphs. Instead, they simply add a line break, which produces a `<br/>` in HTML. This is identical to how single newlines inside a top-level Markua paragraph function.

Note that any headings inside blockquotes do not show up in the Table of Contents. They can also be formatted differently by Markua Processors.

Asides (`A> `)

Since Markua is for writing books and documents, including technical books and documents, it needs not just a syntax for blockquotes–it also needs a syntax for asides, as well as for blurbs and callouts. These syntaxes are very similar to the Markua syntax for blockquotes. Like blockquotes, any headings inside asides, blurbs and callouts do not show up in the Table of Contents, and they can also be formatted differently by Markua Processors.

We will consider asides first.

Asides are typically short or long notes in books which are tangential to the main idea–sort of like footnotes, but in the body text itself. In technical books, quite often they are formatted in a box. Asides can span multiple pages.

The syntaxes for asides are very similar to blockquotes:

1. By prefacing lines with `A> `, i.e. an A, then a greater than character, then a space.
2. By wrapping the aside in `{aside}` … `{/aside}`

Option #1 is preferable for short asides; option #2 is easier on authors for really long asides.

Like blockquotes, asides can be siblings of paragraphs or nested in them.

Here’s a short aside:

```
A> This is a short aside.
```

Here’s a longer aside, which also contains a heading:

```
A> # A Longer Aside
A>
A> This is a longer aside.
A>
A> It can have multiple paragraphs.
A>
A> The `A> ` stuff can get tedious after a while.
A>
A> This is why the `{aside}` syntax exists.
```

Here's a longer aside using the `{aside}` syntax, which also contains a heading:

```
{aside}
## A Note About Asides

This is a longer aside.

It can have multiple paragraphs.

Asides can also have headings, like this one does.

Multi-paragraph asides are more pleasant using this syntax.
{/aside}
```

Do not confuse the syntax for asides, blurbs and callouts with an attribute list. They are not attribute lists.

Blurbs (`B>`)

Blurbs are like asides, but shorter. A blurb cannot span multiple pages. A blurb can contain headings.

The syntaxes for blurbs are very similar to asides:

1. By prefacing lines with `B> `, i.e. a B, then a greater than character, then a space.
2. By wrapping the blurb in `{blurb}` … `{/blurb}`

Examples:

```
B> This is a short blurb.
```

```
B> # A Longer Blurb
B>
B> This is a longer blurb.
B>
B> It can have multiple paragraphs.
```

```
{blurb}
#A Longer Blurb

This is a longer blurb.

It can have multiple paragraphs.
{/blurb}
```

Supported Attributes for Blurbs

Blurbs also have support for an attribute list, which can contain a `class` attribute as well as other implementation-specific "extension attributes".

Blurb `class` Types

Markua has its origin in Leanpub Flavoured Markdown, and Leanpub has its origin in authoring computer programming books.

In computer programming books, there are a number of blurb types which are a defacto standard:

- `discussion`
- `error`
- `exercise`
- `information`
- `question`
- `tip`
- `warning`

These blurb types can be set on a blurb as its `class` attribute. A Markua Processor can optionally style these blurbs appropriately based on the class, for example by adding custom icons for each class of blurb.

Here's how this looks with the `B>` syntax:

```
{class: warning}
B> This is a warning!
```

Here's how this looks with the `{blurb}` syntax:

```
{class: warning}
{blurb}
This is a warning!
{/blurb}
```

The attribute list must precede the `{blurb}` with no blank line between it.

Syntactic Sugar for Blurb Classes: `D>`, `E>`, `I>`, `Q>`, `T>`, `W>`, `X>`

Having to constantly type `{class: warning}` in a computer programming book with a number of warnings would get tedious, as would any of the other blurb classes listed above.

So, Markua defines a standard shorthand syntax for these classes of blurbs. With this syntax, you use a different letter than `B` in the `B>`, to create a blurb with the appropriate `class`.

These are the syntactic sugar values you can use:

Sugar	Equivalent To a B> With
`D>`	`{class: discussion}`
`E>`	`{class: error}`
`I>`	`{class: information}`
`Q>`	`{class: question}`
`T>`	`{class: tip}`
`W>`	`{class: warning}`
`X>`	`{class: exercise}`

Examples:

```
D> This is a discussion blurb.

E> This is an error blurb.

I> This is an information blurb.

Q> This is a question blurb.

T> This is a tip blurb.

W> This is a warning blurb.

X> This is an exercise blurb.
```

These are equivalent to:

```
{class: discussion}
This is a discussion blurb.

{class: error}
This is an error blurb.

{class: information}
This is an information blurb.

{class: question}
This is a question blurb.

{class: tip}
This is a tip blurb.

{class: warning}
This is a warning blurb.

{class: exercise}
This is an exercise blurb.
```

Note that nothing in this section defines what a Markua Processor must *do* with the given class of blurb. A Markua Processor is free to ignore the `class` entirely. Or it can do something it considers appropriate–Leanpub, for example, uses it to add an appropriate icon from Font Awesome at the left of the blurb.

Using Extension Attributes on Blurbs to add `icon` Support

Markua Processors must ignore any attributes which they do not understand.

Because of this, Markua attribute lists can contain any number of extension attributes. An extension attribute is an attribute which is not defined in the Markua specification.

The `icon` attribute on blurbs is an example of extension attributes.

Markua does not specify that a blurb must support an `icon` attribute, or what it would mean if it did. However, Leanpub understands an `icon` attribute to reference an icon from Font Awesome. The value of this attribute is assumed to be the name of an icon in Font Awesome, without the `fa-` prefix. So, in Leanpub, you can do this:

```
{icon: car}
B> You can't spell carbon without it!

{icon: "warning"}
B> Yes, we're in Font Awesome!

{icon: github}
B> So is GitHub, of course. Unicorns.
```

In Leanpub, this will produce a nice icon of a car, using the Font Awesome icon. In a Markua implementation that does not understand the icon attribute, nothing will be generated for that attribute – it will be functionally equivalent to the attribute not being present.

Let's see whether the Markua processor that produced this document understands the `icon` attribute:

Car. You can't spell carbon without it!

Leanpub. Yes, we're in Font Awesome!

So is GitHub, of course. Unicorns.

TODO_LEANPUB - upgrade Font Awesome version so this is true for Leanpub!

Callouts (C>)

Callouts are essentially blurbs that float.

The main difference between callouts and blubs or asides is that callouts can be freely and creatively positioned by the Markua Processor near the position in the manuscript it was defined. Asides and blurbs must be positioned exactly at the manuscript position. A callout can have page breaks in it, but shouldn't if possible.

The other difference between callouts and blurbs is that like asides, callouts don't support `class` or `icon` (or any) attributes.

The syntaxes for callouts are very similar to blurbs:

1. By prefacing lines with `C> `, i.e. a C, then a greater than character, then a space.
2. By wrapping the callout in `{callout}` ... `{/callout}`

Examples:

```
C> This is a short callout.
```

```
C> # A Longer Callout
C>
C> This is a longer callout.
C>
C> It can have multiple paragraphs.
```

```
{callout}
#A Longer Callout

This is a longer callout.

It can have multiple paragraphs.
{/callout}
```

Span Elements

We've already seen many examples of span elements in this specification, especially in the resources section showing resources inserted as spans. These are the rest of them…

Links

Markua's hyperlink support is a subset of that of Markdown. The **inline** link syntax is supported, as is the **automatic link** shortcut. The reference link syntax and the implicit link name shortcuts are **NOT** supported.

Markdown is a way of writing HTML designed by bloggers, and links are so plentiful in that medium that it makes some sense to support four syntaxes to create links. In ebooks, however, links are not as prevalent, so it makes sense to be opinionated here. Furthermore, in something as large as a book, the potential for id collisions in link definitions using the reference link syntax is a lot higher than in a blog post.

Inline Links

The normal way to create a link is as follows:

```
[link text](absolute_url)
```

In Markdown, the URL can be either absolute or relative, since relative URLs can make sense on web servers. In Markua, however, all URLs must be absolute.

Example:

```
Markua was developed at [Leanpub](http://leanpub.com).
```

Automatic Links

To create a link where the text displayed for the link text is the URL itself, the automatic link syntax can be used. In this syntax, an absolute URL is enclosed in angle brackets.

```
Some text <absolute_url> some text.
```

Example:

```
Markua was developed at <http://leanpub.com>.
```

Explicitly Creating Spans with `[...]`

Surrounding text in square brackets can be useful not just for giving it a URL to link to. If you wish to add attributes to an arbitrary span of text, you can create an arbitrary span of text using square brackets and then add an attribute list immediately afterward:

```
My friend said [bonjour]{lang: fr} to me.
```

With the attribute I have indicated that the bonjour is a French word. So, if the Markua processor uses a different font for French, it can do so.

This is a contrived example, but it makes a lot more sense with intermixing right-to-left languages like Hebrew and Arabic with left-to-right languages–in those cases, bad output can result if a Markua Processor does not realize it needs to reverse directions and switch fonts.

Sometimes a Square Bracket is Just a Square Bracket

If there are no round brackets or curly braces immediately after some text in square brackets, the text in square brackets is just that: text in square brackets. In this case, the square brackets are output as normal text.

This is useful when you want to [sic.] something, etc.

Footnotes

Books and documents often have footnotes.

To add a footnote, you insert a footnote tag using square brackets, a caret and the tag, like this:

```
This has a footnote[^thenote].
```

Then, you define the footnote later in the document, using the same square brackets, caret and tag, followed by a colon, a space and the footnote definition:

```
[^thenote]: This is the footnote content.
```

If you wish to write multiple paragraphs in the footnote, you must indent the subsequent paragraphs by four spaces or one tab:

```
This has a footnote[^thenote].

Here is some unrelated text.

[^thenote]: This is the first paragraph of footnote content.

    This is the second paragraph of footnote content.

Here is some more unrelated text.
```

Endnotes

Books and documents often have endnotes as well. Sometimes these are instead of footnotes, but other times, these are in addition to footnotes. So, it makes sense for Markua to define separate syntaxes for both, rather than just defining one “footnote or endnote” syntax and letting the author pick whether the notes were footnotes or endnotes.

To add an endnote, you insert an endnote tag using square brackets, two carets and the tag, like this:

```
This has an endnote[^^thenote].
```

Then, you define the endnote later in the document, using the same square brackets, two carets and tag, followed by a colon, a space and the endnote definition:

```
[^^thenote]: This is the endnote content.
```

Just as with a footnote, if you wish to write multiple paragraphs in the endnote, you must indent the subsequent paragraphs by four spaces or one tab.

Crosslinks and ids

There are two parts to making a crosslink.

1. Define an id.
2. Reference that id with a crosslink.

Defining an id

There are two ways to define an id:

1. Using an id attribute `{id: some-id}` in an attribute list which can contain other attributes.
2. Using a shorter “syntactic sugar” approach when the id is by itself: `{#some-id}`

The shorter approach is preferred. The main reason the `{id: some-id}` syntax is supported is that it allows it to be added in with other attributes in a larger attribute list. You **CANNOT** add the `{#some-id}` to an attribute list.

In terms of the value of an `id`, it has some special restrictions:

1. The first character in the `id` has to be a lowercase or uppercase letter, i.e. `[a-zA-Z]` if you think in regular expressions.
2. The remaining characters in the `id` have to be a lowercase or uppercase letter or a digit or a hyphen (-) or an underscore (_).
3. You can only define an `id` value once in an entire Markua document, even one that is split over multiple files.

These restrictions ensure that your `ids` can then be linked to by a crosslink from anywhere in the Markua document.

The id needs to be defined on either a block or span element.

Defining an id on a Block Element

To define an id on a block element like a paragraph, figure, heading or even a definition list item, you simply stick the id definition on a line above the start of the block element.

Here's how to use the attribute list syntax to define an id attribute:

```
{id: some-id}
This is a paragraph with the id of `some-id`.
```

Here's how to use the shorter "syntactic sugar" approach to define an id attribute:

```
{#some-id}
This is a paragraph with the id of `some-id`.
```

Defining an id on a Span Element

To define an id on a span element you simply add the id definition immediately after the span element.

Here's how to use the attribute list syntax to define an id attribute on a span elemnet:

```
The word Markua{id: markua} has an id.

Leanpub is based in **Victoria, BC, Canada**{#victoria}.
```

Here's how to use the shorter "syntactic sugar" approach to define an id attribute:

```
The word Markua{#markua} has an id.

Leanpub is based in **Victoria, BC, Canada**{id: victoria}.
```

Referencing an `id` With a Crosslink

Regardless of how you defined the id, you then link to it to create a crosslink. To do this, you use the `#` character and the id in a link:

```
[link text](#some-id)
```

This syntax is intended to be reminiscent of HTML anchor tags, not of `h1` titles in Markua.

Note that order of definition and use does not matter: crosslinks will work regardless of whether the `id` is defined before or after the use of it.

Rules for `id`s and Crosslinks

- If a Markua document contains duplicate `id` attribute values, the **first** one is used and the subsequent ones are ignored. A Markua Processor should output a warning about duplicate `id`s.
- Crosslinks that reference an unused `id` may either be created as a (broken, non-functional) link or be created as normal text (not a link) by a Markua Processor. The Markua Processor may also output a warning about this somewhere, but not in the actual document text itself.

Referencing Chapter, Section and Figure Heading Names and Numbers in Crosslinks

Chapters, sections and figures with captions often have two useful properties for writers:

1. A name which is often short and useful to reference.
2. A number, if numbering is turned on.

Whether the numbers exist is determined by the `number-chapters`, `number-parts` and `number-figures` attributes.

Here's how these references to titles and numbers work:

- `#t` is for "title"
- `#n` is for "number"
- `#d` is for "description" (e.g. "Figure", "Chapter", "Section")
- `#f` is for "full title"

So, for "Figure 8.2: Anatomy of a Squirrel", these are:

- `#t` is "Anatomy of a Squirrel"
- `#n` is "8.2"
- `#d` is "Figure"
- `#f` is "Figure 8.2: Anatomy of a Squirrel"

Note that in this example, "Anatomy of a Squirrel" was typed by the author, whereas "Figure 8.2: " was generated by the Markua Processor. It does not matter; both can be referenced.

Also, note that regardless of section level, sections referenced in `#d` or `#f` are all called "Section" (not "Sub-Section", "Sub-Sub-Section", etc.)

The expectation is that `#f` will be used by authors who don't mind verbosity, and #t and #n will be used by authors who prefer control and brevity. The #d is for very lazy authors who like saving keystrokes and/or who don't know whether their publisher will call the code samples "Listing", "Example" or some other word and want to protect themselves against extra work.

The `code-sample-names`, 'figure-names `and` table-names' settings control the words used to name things.

Examples:

```
This is discussed in [section #n, #t](#crosslinks).

This is discussed in [#f](#crosslinks).

This is discussed in a [#d](#crosslinks) above.

See [chapter #n](#span-elements), which is the best chapter in this book.

This is in figure [#n](#fancy-diagram), arguably the fanciest diagram in this document.
```

Notes:

- Figures, chapters, code samples, etc all have implicit numbering. So, #n always works even if numbering is off. However, you will confuse readers if you refer to numbering they cannot see.
- If numbering is off, the `#f` will not include either the `#d` or `#n` parts: it will be "Anatomy of a Squirrel" not "Figure 8.2: Anatomy of a Squirrel".

Scene Breaks

In fiction, scene breaks are sometimes added between paragraphs in a chapter to denote a break in context.

To make a scene break, create a line which has only hyphens, asterisks or underscores on it, and which has at least three of them. There may be whitespace between the hyphens or asterisks. Scene breaks must have a blank line above them and below them.

Example:

```
This is before the first scene break.

* * *

This is after the first scene break and before the second.

---

This is after the second scene break.
```

Markua Processors can take great liberties in how they output scene breaks. For example, scene breaks can be displayed as a few centered asterisks, an image, or just a blank line, depending on whether the book is fiction or another type of book. Scene breaks can also be displayed as the `hr` element in HTML, but that is not a requirement.

Soft hyphen

In certain rare situations, it's desirable to be able to give hints to a Markua Processor about where hypenation can be done.

This attribute makes us sad. However, it's a nod to reality. Hopefully this attribute is never used by authors writing the first draft of anything.

Soft hyphens are not output in HTML.

To insert a soft hyphen, used a backslash followed by a hyphen:

Example:

```
My name is Rumpel\-stiltskin.
```

Metadata

Attributes

Attributes are used to do everything from specify the language of code blocks, add ids for crosslinkng and even support extensions to Markua. We have already seen attributes throughout the specification in the attribute lists we have encountered.

Attribute List Format

An attribute list is one or more key-value, comma-separated pairs:

`{key_one: value1, key_two: value_two, key_three: "value three!", key_four: true, key_five: 0, key_six: 3.14}`

Note that you can skip the space between the colon and the value: the following `{format: ruby}` and `{format:ruby}` both work. However, for consistency I recommend always adding a space.

An attribute list can be inserted into a Markua document in one of three ways:

1. Immediately above a block element (e.g. heading, figure, etc), with one newline separating it from the block element.
2. Immediately after a span element (e.g. a word, italicized phrase, etc), with no spaces separating it from the span element.
3. On a line by itself, with one blank line above and below it. In this format, the attribute list contains directives.

Regarding #2 and #3: Any line outside of a code resource which starts with an opening curly brace { and ends with a closing curly brace } is assumed to be an attribute list, and will not be output by a Markua processor. If you want to start a line with a literal opening curly brace { you need to preface it with a backslash (\).

Attribute Keys

The keys of attributes must consist exclusively of lowercase letters and underscores (_). (For you programmers, this is `[a-z_]`.)

Attribute Value Types

Attributes are typed as `boolean`, `integer`, `float` or `text`. Most attributes are `text`.

Every attribute in Markua has a default.

Here's how attribute values are interpreted:

`boolean`
The values are `true` or `false` are boolean values. Unless otherwise specified by the attribute itself, the default value of a boolean attribute is `false`. Any value other than `true` is interpreted as `false`.

`integer`
Any number which does not contain a decimal point is interpreted as an integer. Unless otherwise specified by the attribute itself, the default value of an integer is `0`.

`float`
Any number which contains a decimal point is interpreted as a float. If an attribute is a float, it can accept attribute values that are integers as well. (You're not required to type `0.0` for `0`.) Unless otherwise specified by the attribute itself, the default value of a float is `0`.

`text`
Any value other than `true`, `false` or a number is text. A text attribute value must be enclosed in quotes unless every character in it is a lowercase letter, uppercase letter, number, hyphen (`-`) or underscore (_). (For you programmers, this is `[a-zA-Z0-9_-]`.)

Only text attribute values can be enclosed in quotation marks.

If a text attribute value contains a quote, it must be escaped: e.g. `{caption: "\"Fresh\" Fish"}`

`id` Attributes

As previously discussed, there is special syntactic sugar for ids: `{#foo}` is equivalent to `{id: foo}`. However, ids are just attributes.

Extension Attributes

Markua Processors must silently ignore any attributes which they do not understand. This is true whether the attributes are inserted in an attribute list attached to a span, block or even in free-floating directives.

Because of this, Markua attribute lists can contain any number of extension attributes. An extension attribute is an attribute which is not defined in the Markua specification.

This encourages competition in the Markua ecosystem, while ensuring that Markua implementations do not choke on Markua input which goes beyond their capabilities.

Extension attributes go far beyond adding icons to blurbs: they allow for specialized uses of Markua. Since CSS is so powerful, with creative uses of custom attributes and custom CSS, Markua documents can be transformed. Some obvious uses of extension attributes include adding CSS classes which can then be styled to set fonts, adding custom types to figures for things like lemmas and theorems, etc.

This ensures that new attributes can be added to future versions of Markua without a negative effect on older Markua implementations. It also ensures that new versions of Markua can simply stop supporting attributes defined in this version of Markua without needing to specify anything special.

Index Entries

Markua supports adding index entries via the attribute list syntax. Index entries let authors or indexers produce professional-looking indexes in Markua books.

Index entries can either be attached to block or span elements using the same attribute list syntax. In fact, index entries can just be added as part of a larger attribute list.

The actual syntax of what the value of an index entry looks like is inspired by LaTeX[20].

The key of an index entry is `i`, for index.

In its simplest form, an index entry is simply `{i: "hello"}`. Like any attribute list, you don't need a space between the colon and the quote–you can also do `{i:"hello"}`.

These are the various formats of an index entry:

```
{i: hello}
{i: "hello"}
{i: "Armstrong, Peter"}
{i: "Yahoo!"}
{i: "*hello*"}
{i: "**hello**"}
{i: "hello!Peter"}
{i: "hello!*Peter*"}
{i: "hello!**Peter**"}
{i: "Peter|see{i:'hello'}"}
{i: "Jen|seealso{i:'Jenny'}"}
```

Here's what they do:

`{i: hello}`
: Adds an index entry for `hello`. If an index entry has no punctuation or formatting then it does not need quotes.

`{i: "hello"}`
: Adds an index entry for `hello`. Quotes are always fine to use, even when not required.

`{i: "Armstrong, Peter"}`
: Adds an index entry for `Armstrong, Peter`. The quotes are always omitted. Their function is to allow things like exclamation marks and other punctuation to be added without fear, in case you don't feel like learning which punctuation is safe.

[20]https://en.wikibooks.org/wiki/LaTeX/Indexing

`{i: "Yahoo!"}`
: Adds an index entry for `Yahoo!`. Exclamation marks in index entries are delimiters, so to add a literal exclamation mark, the index entry needs to be in quotes. The other characters in index entries that must be in quotes are `|`, `{` and `}`.

`{i: "*hello*"}`
: Adds an index entry for `hello`, with `hello` in italics.

`{i: "**hello**"}`
: Adds an index entry for `hello`, with `hello` in bold.

`{i: "hello!Peter"}`
: Adds an index entry for `Peter` which is a sub-entry of `hello`.

`{i: "hello!*Peter*"}`
: Adds an index entry for `Peter` (with *Peter* in emphasis) which is a sub-entry of `hello`. Note that this cannot be combined with a see or seealso (the | syntax).

`{i: "hello!**Peter**"}`
: Adds an index entry for `Peter` (with **Peter** in strong emphasis) which is a sub-entry of `hello`. Note that this cannot be combined with a see or seealso (the | syntax).

`{i: "Peter|see{i:'hello'}"}`
: Adds an index entry for `Peter`, which references the index entry `hello` with the equivalent of "see" in the language of the book. Note that this cannot be combined with a sub-entry (the ! syntax).

`{i: "Jen|seealso{i:'Jenny'}"}`
: Adds an index entry for `Jen`, which references the index entry `Jenny` with the equivalent of "see also" in the language of the book. Note that this cannot be combined with a sub-entry (the ! syntax).

Index entries are case sensitive. For example, `{i: "mark"}` and `{i: "Mark"}` are distinct entries. (The first is for a result or a smudge, the second is a person's name.)

To attach an index entry to the start of a block, put it on its own line above a block:

```
{i: "hello"}
I just came to say hello, hello, hello, hello
```

To attach an index entry to a word, just add the index entry after the word:

```
I just came to say hello{i: "hello"}, hello, hello, hello
```

To attach an index entry to a span element, just add the index entry after the span element:

```
The first program that a programmer writes in a language is usually *Hello World*{i: "Hello World"}
```

Index entries can have commas and other punctuation (except colons) in their definition:

```
My wife read some book about a whale by Herman Melville{i: "Melville, Herman"}.
```

Multiple index entries can exist in a block, or even a sentence:
Supposedly the great-great-great-granduncle of the musician Moby{i: "Moby"} was Herman Melville{i: "Melville, Herman"}, the author of a book about a whale{i: "Moby-Dick; or, The Whale"}.

Note that adding index entries is best left until the end of a book. At that time, ids like {#myid} can be converted to {id: #myid, i: "blah"} if index entries are being added where ids already are.

Directives

Directives are switches which affect the future behaviour of a Markua Processor.

The syntax for directives is simple: they are just contained in an attribute list. The only difference is that the attribute list is inserted an a line by itself, with one blank line above and below it.

Language Attributes and Directives

A Markua document has a language. By default, the language is `en` for English. The default language can be set in the settings, discussed later.

Every human language has a direction, either left-to-right or right-to-left. Also, some fonts only handle certain languages correctly.

So, it's important to be able to indicate in Markua which language the text is in, so that a Markua Processor can use the correct fonts and apply the correct direction.

To do this, you use the `lang` attribute.

If you use the `lang` attribute in an attribute list of a block or span, the `lang` change just applies to that block or span, which can help ensure that the font was corectly set.

However, if you use the `lang` attribute as a directive, it applies as the new default until a different `lang` directive is encountered.

The following example shows how you would say hello in English, then switch the language to Hebrew, then Chinese (Simplified), then Arabic, and then switch back to English to say goodbye:

```
hello

{lang: he}

Hebrew

text

here

{lang: zh}

Chinese
```

```
text

here

{lang: ar}

Arabic

text

here

{lang: en}

goodbye
```

Note that I didn’t actually say hello or goodbye in Hebrew, Chinese or Arabic because the code font used for this specification does not support them all.

> In Leanpub Flavoured Markdown, we had `{rtl}` and `{ltr}` directives. These are not supported in Markua, and neither is a `{dir}` attribute in general: languages only have one direction, so there’s no need to have separate `dir` and `lang` attributes.

Book `section` Directives

Most published books have three types of material in them: the front matter, the text and the back matter.

What authors write, the manuscript, is typically what goes into the text, or main matter, of the book. In style guidelines such as the Chicago Manual of Style this is called the text; in formats such as TeX and LaTeX it is called main matter.) It’s what is numbered with Arabic numerals starting from 1.

There’s a bunch of other stuff (the Dedication, Epigraph, Table of Contents (or ToC), Foreword, Preface, etc.) which can come before the main text of the book. This is called “front matter”. Some of the front matter comes before the ToC and is not numbered; the rest of the front matter that comes after the ToC gets numbered with Roman numerals.

There’s also a bunch of other stuff (acknowledgments, appendices, the index, etc) which can come after the main matter. This is called the “back matter”.

If Markua just relied on its headings support there would be no good way to accomplish the division of a manuscript into front matter, main matter and back matter. (We could try some convention about heading names, but that would be a highly objectionable, English-centric hack.) Furthermore, there would be no good way to determine which parts of the front matter got numbered and which did not. Also, there would be no good way to give explicit hints to Markua processors about how to style the various parts of a book.

The book `section` directive is the solution to this.

Adding multiple instances of the `section` directive throughout the book lets you indicate to Markua Processors what section of a book they are in, in order to affect numbering, formatting and other concerns like the placement of the Table of Contents.

Markua is a fairly minimalist format, so this is all accomplished with one directive. Furthermore, for authors who do not know about this directive, nothing bad or unexpected will happen.

There are three types of values of the book `section` directive: front matter values, the `mainmatter` value and back matter values. These values are named for the different book sections identified in the Chicago Manual of Style.

The following values of the `section` directive can occur in front matter:

- `half-title`
- `series-title`
- `title-page`
- `copyright`
- `dedication`
- `epigraph`
- `toc`
- `figures`
- `tables`
- `foreword`
- `preface`
- `acknowledgments` (note that this value can also typically occur after the `{section: mainmatter}` book section directive)
- `introduction`
- `abbr` (yes, it's called `abbr` and not `abbreviations`, and note that this can also typically occur after the `{section: mainmatter}` book section directive)
- `chronology` (note that this can also typically occur after the `{section: mainmatter}` book section directive)

The `{section: mainmatter}` value indicates that the front matter is over. It is the start of the first page of the text.

The following values of the `section` directive can occur in back matter:

- `appendices`
- `notes`
- `glossary`
- `bibliography` (note that a Markua processor may wish to format the text inside this section appropriately)
- `contributors`
- `illustration-credits`
- `index`

You can write a book or document in Markua without using any directives at all, including book `section` directives. However, if you add a `section` directive with a value of any of the front matter values, the `{section: mainmatter}` directive is required.

If you're an author and this seems confusing, don't fear. Almost all of the directives will be added either by someone working for a publisher or automatically by book generation software. The only directives that a typical author will use are `{section: dedication}` and `{section: mainmatter}`.

Also, note that Markua Processors may ignore all of the book `section` directives. Book `section` directives are only hints that indicate that the Markua Processor may wish to do things like switch to hanging indents, change the font size, use a custom layout etc based on what section of the book they are in.

While these book `section` directives are mere hints, there are strict rules about their use:

1. Like any directive, each book `section` directive must be on a line by itself with blank lines above and below it.
2. Each value of a book `section` directive can only appear once.
3. The book `section` directives are all optional; it is legal to have a Markua document with no book `section` directives.
4. If any book section directives with values which occur in front matter are present, a `{section: mainmatter}` directive is required.

Authors new to Markua can completely ignore all directives, including book `section` directives. Only when they wonder how to make certain sections excluded from the numbering, or to have Roman numerals etc, do they have to worry about book `section` directives. And using directives is very simple, despite how formally specified this section makes them.

Settings

As discussed earlier, a Markua document can be written in one file or multiple manuscript files. If a manuscript is written in multiple files, these files are concatenated together by the Markua processor to produce one temporary manuscript file, and that one file is used as the input.

The settings for a Markua document can alse be provided in four ways:

1. at the top of the first file, enclosed in curly braces
2. in a separate settings file
3. separately (e.g. via a web interface)
4. some combination of `#1`-`#3`

Typically approaches `#2`, `#3` and `#4` are used when a Markua document is being written in multiple manuscript files.

With approach `#1`, the settings are added at the top of the file:

```
{
title: Markua Specification
authors: Peter Armstrong
}

## The Magical Typewriter

Imagine...
```

With approaches #2, #3 and #4, the Markua Processor must act as though there is one file as follows:

```
{
(the contents of the actual or synthesized settings file)
}
(the contents of the first manuscript file)

(the contents of the second manuscript file)
...
```

The settings contain information about the document. This will include things like the title, subtitle, copyright, author names, etc.

Settings Are Just Newline-Separated Key-Value Pairs

As the title of this section states, settings are just newline-separated key-value pairs.

Here's an example of the settings metadata:

```
title: Markua Specification
authors: Peter Armstrong
contributors: Scott Patten, Braden Simpson and Jordan Ell
copyright: Peter Armstrong
```

The format of the metadata is one entry per line, with the keys and values separated by a colon and a space.

Any whitespace at the beginning or end of the keys and values will be stripped. Only the first colon is important. Subsequent colons on a line will just be part of the value.

Settings Keys and Values

The following keys can be used in the settings. None of theme are mandatory, although you will almost certainly want to provide a value for `authors`, `copyright` and `title`.

authors
: The authors of the Markua document. The default is "Anonymous".

code The language that code is. The default is `guess`, which means to guess at the code language based on the syntax encountered (or the file extension for external code samples), and attempt to syntax highlight appropriately. A good alternative is `text`, which means no syntax highlighting should be used, but the code should be in a monospaced font suitable for a programming language. Besides these options, you can specify a particular programming language used, such as `ruby` or `java`. If a Markua processor does not recognize the programming language specified, it must format it as `text`.

code-sample-names
What to call code samples. The default is "Example".

copyright
The text of the copyright holder. The default is "Anonymous".

direction
`rtl` or `ltr`. The default is `ltr`.

figure-names
What to call figures. The default is "Figure".

filename
The filename (minus extension) of the generated output file(s). The default is "untitled".

language
The IS0 639-2 alpha-3 language code of the language that the Markua document is written in. The list of choices is here: http://www.loc.gov/standards/iso639-2/php/code_list.php. The default is `eng`.

license
The copyright license used. The choices are `standard`, `public-domain`, `cc-by`, `cc-by-sa`, `cc-by-nd`, `cc-by-nc`, `cc-by-nc-sa` and `cc-by-nc-nd`. The `standard` choice means standard copyright. The `public-domain` means that the work is considered to be in the public domain. The other choices are Creative Commons licenses, specified here: https://creativecommons.org/licenses/. The default value is `standard`.

number-chapters
This attribute sets how chapter numbering is done. The values are `off`, `chapter`, `section`, `sub-section`, `sub-sub-section` and `all`. The default is `chapter`, which means to number chapters but not sections or below.

number-parts
This attribute sets whether parts are numbered, if present. Legal values are `true` or `false`. The default value is `false`. Any value other than `true` is interpreted as `false`.

number-figures
This attribute sets whether figures are numbered, if present. The values are `off`, `single` and `double`. The default is `double`. The choice `single` means to number figures in one large list (Figure 1, Figure 2, etc.). The choice `double` means to number by chapter, e.g. `Figure 8.1` for the first figure in chapter 8. Per Chicago, figures in an Appendix would be listed as `Figure A.1` for Appendix A, etc.

number-code-samples
: This attribute sets whether code sample code samples are numbered, if present. The values are `off`, `single` and `double`. The default is `double`. The choice `single` means to number code samples in one large list (Example 1, Example 2, etc.). The choice `double` means to number by chapter, e.g. `Example 8.1` for the first code sample in chapter 8. Per Chicago, code samples in an Appendix would be listed as `Example A.1` for Appendix A, etc.

number-tables
: This attribute sets whether tables are numbered, if present. The values are `off`, `single` and `double`. The default is `double`. The choice `single` means to number tables in one large list (Table 1, Table 2, etc.). The choice `double` means to number by chapter, e.g. `Table 8.1` for the first table in chapter 8. Per Chicago, tables in an Appendix would be listed as `Table A.1` for Appendix A, etc.

subtitle
: The subtitle of the Markua document. The default is blank (an empty string, in programming terms).

table-names
: What to call tables. The default is “Table”.

title
: The title of the Markua document. The default is “Untitled”.

urls
: The way that URLs should be displayed when used as links like `[Markua](http://markua.com)`. The choices are `none`, `inline` and `footnote`. The default is `footnote`, in which the URL is made as a footnote (positioned at the bottom of the page, chapter or book per the value of the `footnotes` attribute) which is a clickable link. Note that angle-bracketed inline URLs like http://markua.com never create a footnote.

Escaping Special Characters

Curly Braces (`{` and `}`)

Curly braces are special in Markua. At the beginning of a line, an opening curly brace (`{`) starts an attribute list. In the middle of a block element, an opening curly brace starts an index entry. And, at the top of a manuscript.txt file (if the single file approach is used), an opening curly brace starts a settings block. So, to use a curly brace as an actual curly brace character, you need to backslash-escape it like this: `\{`. (Note that this does not apply inside code or other resources: Markua does not process anything inside them.)

Backticks (‘)

A backtick in text content (such as this paragraph) starts an inline span resource such as a code resource.

So, to show a literal backtick in a span, you start and end it with *two* backticks.

This is the same reason that there are two ways to create inline figures, with three backticks and three tildes. Starting an inline figure with three tildes lets three backticks be used without closing the figure; similarly, starting an inline figure with three backticks lets three tildes be used without closing the figure. This is necessary for Markua to be usable to write about Markua!

CriticMarkup is Supported in Markua

CriticMarkup[21] describes itself as follows:

> a way for authors and editors to track changes to documents in plain text. As with Markdown, small groups of distinctive characters allow you to highlight insertions, deletions, substitutions and comments, all without the overhead of heavy, proprietary office suites.

Basically, supporting CriticMarkup gives Markua authors the benefit of track changes in Word, but with all the benefits of using Markua. An editor can use CriticMarkup to make suggestions about text to add, delete or replace, as well as leaving comments and highlights on selected text.

The largest benefit to using the CriticMarkup syntax in Markua is that since Markua documents are in plain text, you can use proper version control (like Git or Mercurial) with them, and this includes having the comments versioned right along with the rest of the Markua manuscript. Finding a comment that you deleted six months ago is as easy as checking out the revision that contained the comment.

If you enjoy using track changes in Word, then you'll enjoy using CriticMarkup. If you don't enjoy working with editors or getting feedback about your writing in the text itself, then CriticMarkup isn't for you! If this is the case, don't worry about it: the support for CriticMarkup has no impact on the rest of Markua. If you don't want to use it, you don't need to learn about it.

There are five types of changes supported by the markup in CriticMarkup:

Change	**Markup**
Addition	{++ … ++}
Deletion	{-- … --}
Substitution	{~~ … ~> … ~~}
Comment	{>> … <<}
Highlight	{== … ==}{>> … <<}

These types of markup do not conflict with any Markua markup for attribute lists, since the opening curly brace is always followed by special characters.

Markua Processors **must** support CriticMarkup by outputting the additions, deletions, substitutions, comments and highlights appropriately. The HTML Mapping chapter contains some examples of this. To learn more about how to use CriticMarkup, read the very well-written CriticMarkup spec[22].

[21]http://criticmarkup.com/
[22]http://criticmarkup.com/spec.php

Markua HTML Mapping

TODO - The rest of this document needs to be rewritten. Please ignore everything after this point.

Poetry

Single Newlines Inside a Paragraph is a Line Break

Markua Syntax

```
I grant I never saw a goddess go;
My mistress when she walks treads on the ground.
     And yet, by heaven, I think my love as rare
     As any she belied with false compare.
```

HTML Output

<p>
I grant I never saw a goddess go;<br/>
My mistress when she walks treads on the ground.<br/>
And yet, by heaven, I think my love as rare<br/>
As any she belied with false compare.<br/>
</p>

Resources

Span Resources

Example 1: Span Images with Alt Text

Markua Syntax

```
Markua has a fancy ![Markua two trees logo](markua-logo-small.png) logo!

Markua also has a fancy ![Markua two trees icon](https://pbs.twimg.com/profile_images/525513829455650817/mIba4\
bYL_400x400.png) Twitter avatar!
```

HTML Output

```
<p>Markua has a fancy <img src="resources/markua-logo-small.png" alt="Markua two trees logo"/> logo!</p>
<p>Markua also has a fancy <img src="https://pbs.twimg.com/profile_images/525513829455650817/mIba4bYL_400x400.\
png" alt="Markua two trees icon"/> Twitter avatar!</p>
```

Example 2: Span Images with no Alt Text

Markua Syntax

```
Markua has a fancy ![](markua-logo-small.png) logo!
```

HTML Output

```
<p>Markua has a fancy <img src="resources/markua-logo-small.png"/> logo!</p>
```

Figures

Example 1: Figures with Alt Text and a Caption

Markua Syntax

```
Markua has a fancy logo:

![the word Markua with the two asterisk trees beside it](markua-logo.png "The Markua Logo")

Markua also has a fancy Twitter avatar:

![Markua two trees icon](https://pbs.twimg.com/profile_images/525513829455650817/mIba4bYL_400x400.png "The Mar\
kua Icon")

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class="figure">
  <img src="media/markua-logo.png" alt="the word Markua with the two asterisk trees beside it"/>
  <p class="caption">The Markua Logo</p>
</div>
<p>Markua also has a fancy Twitter avatar:</p>
<div class="figure">
  <img src="https://pbs.twimg.com/profile_images/525513829455650817/mIba4bYL_400x400.png" alt="Markua two tree\
s icon"/>
  <p class="caption">The Markua Icon</p>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Example 2: Figures with no Alt Text or Caption

Markua Syntax

```
Markua has a fancy logo:

![](markua-logo.png)

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class='figure'>
  <img src="media/markua-logo.png"/>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Example 3: Figures with Alt Text but no Caption

Markua Syntax

```
Markua has a fancy logo:

![the word Markua with the two asterisk trees beside it](markua-logo.png)

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class="figure">
  <img src="media/markua-logo.png" alt="the word Markua with the two asterisk trees beside it"/>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Example 4: Figures with a Caption but no Alt Text

Markua Syntax

```
Markua has a fancy logo:

![](markua-logo.png "The Markua Logo")

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class="figure">
  <img src="media/markua-logo.png"/>
  <p class="caption">The Markua Logo</p>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Example 5: A Figure with `float` Attribute and a Percentage `width` Attribute

Markua Syntax

```
Markua has a fancy logo:

{float: "left", width: "50%"}
![the word Markua with the two asterisk trees beside it](markua-logo.png "The Markua Logo")

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class="figure">
  <img src="markua-logo.png" style="float: left; width: 50%;" alt="the word Markua with the two asterisk trees\
 beside it"/>
  <p class="caption">The Markua Logo</p>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Example 6: A Figure with a `width` Attribute of `fullbleed`

Markua Syntax

```
Markua has a fancy logo:

{width: "fullbleed"}
![the word Markua with the two asterisk trees beside it](markua-logo.png "The Markua Logo")

Thanks Justin, it was fun making the logo with you!
```

HTML Output

```
<p>Markua has a fancy logo:</p>
<div class="figure">
  <img src="markua-logo.png" class="fullbleed" style="width: 100%;" alt="the word Markua with the two asterisk\
 trees beside it"/>
  <p class="caption">The Markua Logo</p>
</div>
<p>Thanks Justin, it was fun making the logo with you!</p>
```

Video Resources

Audio Resources

Code Resources

Code Spans

Example

Markua Syntax

```
Here is some `hello world` span code.
```

HTML Output

```
<p>Here is some <code>hello world</code> span code.</p>
```

Example

Markua Syntax

```
Here is some ``code with a backtick (`) in it`` and other text.

If you want to have a backtick by itself, just use you need to have spaces around it like this `` ` ``. The en\
closed backtick cannot start or end the code block.
```

HTML Output

```
<p>Here is some <code>code with a backtick (`) in it</code> and other text.</p>
<p>If you want to have a backtick by itself, just use you need to have spaces around it like this <code>`</cod\
e>. The enclosed backtick cannot start or end the code block.</p>
```

Paragraphs

Markua Syntax

```
This is a paragraph. The number of sentences does not matter.

This is another paragraph.

This is a third paragraph.
```

HTML Output

```
<p>This is a paragraph. The number of sentences does not matter.</p>
<p>This is another paragraph.</p>
<p>This is a third paragraph.</p>
```

Lists

Examples

Example 1: A list (asterisks, one space after each)

Markua Syntax

```
lorem

* foo
* bar
* baz

ipsum
```

HTML Output

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

Example 2: A list (asterisks, two spaces after each)

Markua Syntax

```
lorem

*  foo
*  bar
*  baz

ipsum
```

HTML Output

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

Example 3: A list (asterisks, three spaces after each)

Markua Syntax

```
lorem

*   foo
*   bar
*   baz

ipsum
```

HTML Output

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

Example 4: A list (asterisks, four spaces after each)

Markua Syntax

```
lorem

*    foo
*    bar
*    baz

ipsum
```

HTML Output

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

Example 5: A list (asterisks, one tab after each)

Markua Syntax

```
lorem

* foo
* bar
* baz

ipsum
```

HTML Output

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

Example 6: A list (hyphens, one space after each)

Markua Syntax

```
lorem

- foo
- bar
- baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ul class="hyphen">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ul>
<p>ipsum</p>
```

Example 7: A list (pluses, one space after each)

Markua Syntax

```
lorem

+ foo
+ bar
+ baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ul class="plus">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ul>
<p>ipsum</p>
```

Example 8: A list (one element list with asterisk)

Markua Syntax

```
lorem

* foo

ipsum
```

HTML Output

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

Example 9: A list (one element list with hyphen)

Markua Syntax

```
lorem

- foo

ipsum
```

HTML Output

```
<p>lorem</p>
<ul class="hyphen">
  <li>foo</li>
</ul>
<p>ipsum</p>
```

Example 10: A list (one element list with plus)

Markua Syntax

```
lorem

+ foo

ipsum
```

HTML Output

```
<p>lorem</p>
<ul class="plus">
  <li>foo</li>
</ul>
<p>ipsum</p>
```

Example 11: Not a list (variable number of spaces after bullets)

Markua Syntax

```
lorem

*    foo
*   bar
*  baz

ipsum
```

HTML Output

```
<p>lorem</p>
<p>*    foo<br/>
*   bar<br/>
*  baz</p>
<p>ipsum</p>
```

Ordered Lists (i.e. Numbered Lists)

Examples: Decimal Numbers

Example 1

Markua Syntax

```
lorem

1. foo
2. bar
3. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 2

Markua Syntax

```
lorem

9. foo
10. bar
11. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol start="9">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 3

Markua Syntax

```
lorem

3. foo
2. bar
1. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol start="3" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 4

Markua Syntax

```
lorem

11. foo
10. bar
9. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol start="11" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 5

```
lorem

1. foo
1. bar
1. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Examples: Uppercase Letters

Example 1

Markua Syntax

```
lorem

A. foo
B. bar
C. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="A">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 2

Markua Syntax

```
lorem

I. foo
J. bar
K. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="A" start="9">
<ol start="9">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 3

Markua Syntax

```
lorem

C. foo
B. bar
A. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="A" start="3" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 4

Markua Syntax

```
lorem

K. foo
J. bar
I. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="A" start="11" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 5

```
lorem

A. foo
A. bar
A. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="A">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Examples: Lowercase Letters

Example 1

Markua Syntax

```
lorem

a. foo
b. bar
c. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="a">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 2

Markua Syntax

```
lorem

i. foo
j. bar
k. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="a" start="9">
<ol start="9">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 3

Markua Syntax

```
lorem

c. foo
b. bar
a. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="a" start="3" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 4

Markua Syntax

```
lorem

k. foo
j. bar
i. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="a" start="11" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 5

```
lorem

a. foo
a. bar
a. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="a">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Examples: Uppercase Roman Numerals

Example 1

Markua Syntax

```
lorem

I. foo
II. bar
III. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 2

Markua Syntax

```
lorem

IX. foo
X. bar
XI. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I" start="9">
<ol start="9">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 3

Markua Syntax

```
lorem

III. foo
II. bar
I. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I" start="3" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 4

Markua Syntax

```
lorem

XI. foo
X. bar
IX. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I" start="11" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 5

```
lorem

I. foo
I. bar
I. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Examples: Lowercase Roman Numerals

Example 1

Markua Syntax

```
lorem

i. foo
ii. bar
iii. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="i">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 2

Markua Syntax

```
lorem

ix. foo
x. bar
xi. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="i" start="9">
<ol start="9">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 3

Markua Syntax

```
lorem

iii. foo
ii. bar
i. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="i" start="3" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 4

Markua Syntax

```
lorem

xi. foo
x. bar
ix. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="i" start="11" reversed="true">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

Example 5

```
lorem

i. foo
i. bar
i. baz

ipsum
```

HTML Output

```
<p>lorem</p>
<ol type="I">
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ol>
<p>ipsum</p>
```

In Markua, A Single Element Ordered List is Not a List

Example

Markua Syntax

```
foo

1. This isn't a list.

bar

1975. Also, not a list.

lorem

a. This isn't a list either.

ipsum

A. This isn't a list either.

dolor

i. This isn't a list either.

sit

I. This isn't a list either.

amet
```

HTML Output

```
<p>foo</p>
<p>1. This isn't a list.</p>
<p>bar</p>
<p>1975. Also, not a list.</p>
<p>lorem</p>
<p>a. This isn't a list either.</p>
<p>ipsum</p>
<p>A. This isn't a list either.</p>
<p>dolor
<p>i. This isn't a list either.</p>
<p>sit</p>
<p>I. This isn't a list either.</p>
<p>amet</p>
```

HTML Output

```
<ol type="A" start="4">
<li>foo</li>
<li>bar</li>
</ol>
```

Ordered List Numbering Rules

Example

Markua Syntax

```
This makes an ordered list with decimal numbers (1, 2, 3):

1. one
1. two
1. three

This does not make a list, since the numbers are not all consecutive:

1. foo
2. bar
4. baz

This makes a list (1975, 1976):

1975. An amazing year.
1976. The year after 1975.

This does not make a list:

1975\. An amazing year.
1976\. The year after 1975.

This does not make a list:

1975. An amazing year.
2010. The year Leanpub was launched.

This makes a lowercase alphabetical ordered list (a, b):

a. one
b. two

This makes a lowercase alphabetical ordered list (e, f):

e. five
f. six

This makes a lowercase alphabetical ordered list (i, ii, iii, iv):

i. one
ii. two
iii. three
iv. four

Not a list:

i. one
ii. two
iv. four
```

```
Not a list:

a\. one
b\. two

Not a list:

a. one
c. three

That's it!
```

HTML Output

```
<p>This makes an ordered list with decimal numbers:</p>
<ol>
  <li>one</li>
  <li>two</li>
  <li>three</li>
</ol>
<p>This does not make a list, since the numbers are not all consecutive:</p>
<p>1. foo<br/>
2. bar<br/>
4. baz</p>
<p>This makes a list (1975, 1976):</p>
<ol start="1975">
  <li>An amazing year.</li>
  <li>The year after 1975.</li>
</ol>
<p>This does not make a list:</p>
<p>1975. An amazing year.<br/>
1976. The year after 1975.</p>
<p>This does not make a list:</p>
<p>1975. An amazing year.<br/>
2010. The year Leanpub was launched.</p>
<p>This makes a lowercase alphabetical ordered list (a, b):</p>
<ol type="a">
  <li>one</li>
  <li>two</li>
</ol>
<p>This makes a lowercase alphabetical ordered list (e, f):</p>
<ol type="a" start="5">
  <li>five</li>
  <li>six</li>
</ol>
<p>This makes a lowercase alphabetical ordered list (i, ii, iii, iv):</p>
<ol type="i">
  <li>one</li>
  <li>two</li>
  <li>three</li>
  <li>four</li>
</ol>
<p>Not a list:</p>
<p>i. one<br/>
ii. two<br/>
```

```
iv. four</p>
<p>Not a list:</p>
<p>a\. one<br/>
b\. two</p>
<p>Not a list:</p>
<p>a. one<br/>
c. three</p>
<p>That's it!</p>
```

Headings

```
## Part (h1) #

This is a paragraph.

## Chapter (also h1)

This is a paragraph.

## Section (h2)

This is a paragraph.

### Sub-Section (h3)

This is a paragraph.

#### Sub-Sub-Section (h4)

This is a paragraph.

##### Sub-Sub-Sub-Section (h5)

This is a paragraph.

###### Sub-Sub-Sub-Sub-Section (h6)

This is a paragraph.
```

```
<h1 class="part">Part (h1)</h1>
<p>This is a paragraph.</p>
<h1>Chapter (also h1)</h1>
<p>This is a paragraph.</p>
<h2>Section (h2)</h2>
<p>This is a paragraph.</p>
<h3>Sub-Section (h3)</h3>
<p>This is a paragraph.</p>
<h4>Sub-Sub-Section (h4)</h4>
<p>This is a paragraph.</p>
<h5>Sub-Sub-Sub-Section (h5)</h5>
<p>This is a paragraph.</p>
<h6>Sub-Sub-Sub-Sub-Section (h6)</h6>
<p>This is a paragraph.</p>
```

Text Formatting

Italic Text

Example

Markua Syntax

```
This text _is italic_. This text *is also italic*.
```

HTML Output

```
<p>This text <i>is italic</i>. This text <em>is also italic</em>.</p>
```

Bold Text

Example

Markua Syntax

```
This text __is bold__. This text **is also bold**.
```

HTML Output

```
<p>This text <b>is bold</b>. This text <strong>is also bold</strong>.</p>
```

Bold + Italic Text

Example 1

Markua Syntax

```
This text ___is bold and italic___. This text ***is also bold and italic***.
```

HTML Output

```
<p>This text <b><i>is bold and italic</i></b>. This text <strong><em>is also bold and italic</em></strong>.</p>
```

You can even combine underscores and asterisks. This is not recommended stylistically, but it works.

Example 2

Markua Syntax

```
This text __*is bold and italic*__. This text **_is also bold and italic_**.

This text *__is italic and bold__*. This text _**is also italic and bold**_.
```

HTML Output

```
<p>This text <b><em>is bold and italic</em></b>. This text <strong><i>is also bold and italic</i></strong>.</p>

<p>This text <em><b>is italic and bold</b></em>. This text <i><strong>is also italic and bold</strong></i>.</p>
```

Nesting Bold in Italic, and Italic in Bold

Example

Markua Syntax

```
This text _is italic __and this is nested bold__ and this is still italic_ again.

This text *is italic **and this is nested bold** and this is still italic* again.

This text _is italic **and this is nested bold** and this is still italic_ again.

This text *is italic __and this is nested bold__ and this is still italic* again.

This text __is bold _and this is nested italic_ and this is still bold__ again.

This text **is bold *and this is nested italic* and this is still bold** again.

This text __is bold *and this is nested italic* and this is still bold__ again.

This text **is bold _and this is nested italic_ and this is still bold** again.
```

HTML Output

```
<p>This text <i>is italic <b>and this is nested bold</b> and this is still italic</i> again.</p>

<p>This text <em>is italic <strong>and this is nested bold</strong> and this is still italic</em> again.</p>

<p>This text <i>is italic <strong>and this is nested bold</strong> and this is still italic</i> again.</p>

<p>This text <em>is italic <b>and this is nested bold</b> and this is still italic</em> again.</p>

<p>This text <b>is bold <i>and this is nested italic</i> and this is still bold</b> again.</p>

<p>This text <strong>is bold <em>and this is nested italic</em> and this is still bold</strong> again.</p>

<p>This text <b>is bold <em>and this is nested italic</em> and this is still bold</b> again.</p>

<p>This text <strong>is bold <i>and this is nested italic</i> and this is still bold</strong> again.</p>
```

Underlined Text

Example

Markua Syntax

```
This is ____some underlined____ text.
```

HTML Output

The HTML which is output uses the `<u>` tag, which was added to HTML5.

```
<p>This is some <u>bold and italic</u> text.</p>
```

Strikethrough

Strikethrough is made with `~~two tildes~~` surrounding the text.

Example

Markua Syntax

```
Strikethrough is made with ~~two tildes~~ surrounding the text.
```

HTML Output

```
<p>Strikethrough is made with <span class="strikethrough">two tildes</span> surrounding the text.</p>
```

Markua never specifies CSS, to allow for maximum flexibility and competition on behalf of Markua Processor implementations.

However, in this case, the following CSS rule is strongly recommended:

```
.strikethrough {
  text-decoration: line-through;
}
```

Superscript

To make a span be in superscript, you surround it with carets.

Example

Markua Syntax

```
5^3^ is 125.
```

HTML Output

```
<p>5<sup>3</sup> is 125.</p>
```

Subscript

To make a span be in subscript, you surround it with one tilde each. (This is the same syntax as is used by pandoc.)

Example

Markua Syntax

```
What he thought was H~2~O was H~2~SO~4~.
```

HTML Output

```
<p>What he thought was H<sub>2</sub>O was H<sub>2</sub>SO<sub>4</sub>.
```

ABC: Asides, Blurbs and Callouts

Example

Markua Syntax

```
This is a paragraph before the aside.

A> This is a paragraph in the aside.
A>
A> This is a
A> second paragraph in the aside
A> which contains newlines.
A>
A> This is a third paragraph in the aside.

This is a paragraph after the aside.
```

HTML Output

The HTML output for an Aside is simply a `div`, not a `section` or an `aside`, since an arbitrary aside is not something which is a top-level element of a book. (Also, some legacy EPUB readers would choke on `section` or `aside`.)

TODO - is this correct approach?

```
<p>This is a paragraph before the aside.</p>
<div class="aside">
<p>This is a paragraph in the aside.</p>
<p>This is a<br/>
second paragraph in the aside<br/>
which contains newlines.</p>
<p>This is a third paragraph in the aside.</p>
</div>
<p>This is a paragraph after the aside.</p>
```

Example

Markua Syntax

```
This is a paragraph before the blurb.

B> This is a paragraph in the blurb.
B>
B> This is a
B> second paragraph in the blurb
B> which contains newlines.
B>
B> This is a third paragraph in the blurb.

This is a paragraph after the blurb.
```

HTML Output

The HTML output for an Blurb is simply a `div` with a class of "blurb".

```
<p>This is a paragraph before the blurb.</p>
<div class="blurb">
<p>This is a paragraph in the blurb.</p>
<p>This is a<br/>
second paragraph in the blurb<br/>
which contains newlines.</p>
<p>This is a third paragraph in the blurb.</p>
</div>
<p>This is a paragraph after the blurb.</p>
```

Example

Markua Syntax

```
This is a paragraph before the blurb.

{class: warning}
B> This is a paragraph in the blurb.
B>
B> This is a
B> second paragraph in the blurb
B> which contains newlines.
B>
B> This is a third paragraph in the blurb.

This is a paragraph after the blurb.
```

HTML Output

The HTML output for an Blurb is simply a `div` with a class of “blurb”.

```
<p>This is a paragraph before the blurb.</p>
<div class="blurb warning">
<p>This is a paragraph in the blurb.</p>
<p>This is a<br/>
second paragraph in the blurb<br/>
which contains newlines.</p>
<p>This is a third paragraph in the blurb.</p>
</div>
<p>This is a paragraph after the blurb.</p>
```

Block Quotes

Examples

Example 1: A Block Quote Containing Three Paragaphs

Markua Syntax

```
This is a paragraph before the block quote.

> This is a paragraph in the block quote.
>
> This is a
> second paragraph in the block quote
> which contains newlines.
>
> This is a third paragraph in the block quote.

This is a paragraph after the block quote.
```

HTML Output

The HTML output for a block quote is simply a `div`, not a `section` or an `aside`, since an arbitrary block quote is not something which is a top-level element of a book.

TODO - does `<blockquote>` cause problems for EPUB readers?

```
<p>This is a paragraph before the block quote.</p>
<blockquote>
<p>This is a paragraph in the block quote.</p>
<p>This is a<br/>
second paragraph in the block quote<br/>
which contains newlines.</p>
<p>This is a third paragraph in the block quote.</p>
</blockquote>
<p>This is a paragraph after the block quote.</p>
```

Links

Inline Links

Example

Markua Syntax

```
This is [link text](http://markua.com) a link.
```

HTML Output

```
<p>This is <a href="http://markua.com">link text</a> a link.</p>
```

Automatic Links

Example

Markua Syntax

```
Markua's website is <http://markua.com>.
```

HTML Output

```
<p>Markua's website is <a href="http://markua.com">http://markua.com</a>.</p>
```

Crosslinks and `ids`

Examples

TODO - single or double quotes on id attrs in HTML?

Example 1: Defining an `id` on a single word

Markua Syntax

```
The words foo{#foo} and bar{id: bar} both have ids.
```

HTML Output

```
<p>The words <span id="foo">foo</span> and <span id="bar">bar</span> both have ids.</p>
```

Example 2: Defining an `id` on a code span

Code spans, like any span, can have `id`s.

Markua Syntax

```
Java code is verbose, e.g. `public static void main(String[] args)`{#java}.
```

HTML Output

```
<p>Java code is verbose, e.g. <code id="java">public static void main(String[] args)`</code>.</p>
```

Example 3: Defining an `id` on a span of emphasized text

Any span can have `id`s – even spans which are produced by formatting characters. (Note that in this contrived example, these would be better served by index entries than ids.)

Markua Syntax

```
The ideas behind *Leanpub*{id: leanpub} are explained in Peter's book _Lean Publishing_{#lean}.
```

HTML Output

```
<p>The ideas behind <em id="leanpub">Leanpub</em> are explained in Peter's book <i id="lean">Lean Publishing</\
i>.</p>
```

Example 4: Nesting `id` definitions inside spans

An `id` attribute can be defined on a span or word regardless of whether it is contained in a span.

Markua Syntax

```
This is *a **very very{#very} contrived** example{id: example}*{#contrived} showing this.
```

HTML Output

```
<p>This is <em id="contrived">a <strong>very <span id="very">very</span> contrived</strong> <span id="example"\
>example</span></em> showing this.</p>
```

Scene Breaks

Example

Markua Syntax

```
Yada yada yada.

* * *

Yada yada yada.

- - -

Yada yada yada.

___

Yada yada yada.

**********

Yada yada yada.

----------

Yada yada yada.

____________________

Yada yada yada.
```

HTML Output

```
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
<hr/>
<p>Yada yada yada.</p>
```

Soft hyphen

Markua Syntax

```
You can use soft hyphens to suggest where to break long words like Rumpel\-stiltskin if you wish.
```

HTML Output

<p>You can use soft hyphens to suggest where to break long words like Rumpelstiltskin if you wish.</p>

Definition Lists

Examples

Example 1

Markua Syntax

```
foo

term 1
: definition 1

term 2
: definition 2a
: definition 2b
: definition 2c

term 3
: definition 3

bar
```

HTML Output

```
<p>foo</p>
<dl>
  <dt>term 1</dt>
  <dd>definition 1</dd>
</dl>
<dl>
  <dt>term 2</dt>
  <dd>definition 2a</dd>
  <dd>definition 2b</dd>
  <dd>definition 2c</dd>
</dl>
<dl>
  <dt>term 3</dt>
  <dd>definition 3</dd>
</dl>
<p>bar</p>
```

Embedding Lists, Figures and Code Blocks Inside a Paragraph

Examples

Example 1: Correct List Embedding

Markua Syntax

```
This is a paragraph.

The opinion of the W3C about lists inside paragraphs is:
  * myopic
  * rude
  * amusing
It also:
   1. is not universally accepted
   2. ignores historical precedent
   3. encourages short paragraphs, which is admittedly a plus
This is still part of the second paragraph.

This is a third paragraph.
```

HTML Output

```
<p>This is a paragraph.</p>
<div class="para">The opinion of the W3C about lists inside paragraphs is:
  <ul>
    <li>myopic</li>
    <li>rude</li>
    <li>amusing</li>
  </ul>
  It also:
  <ol>
    <li>is not universally accepted</li>
    <li>ignores historical precedent</li>
    <li>encourages short paragraphs, which is admittedly a plus</li>
  </ol>
  This is still part of the second paragraph.
</div>
<p>This is a third paragraph.</p>
```

Example 2: Incorrect List Embedding

Markua Syntax

```
This is a paragraph.

The opinion of the W3C about lists inside paragraphs is:
* myopic
* rude
* amusing
It also:
1. is not universally accepted
2. ignores historical precedent
3. encourages short paragraphs
This is still part of the second paragraph, but the above lists were not embedded correctly.

This is a third paragraph.
```

HTML Output

```
<p>This is a paragraph.</p>
<p>The opinion of the W3C about lists inside paragraphs is:<br/>
* myopic<br/>
* rude<br/>
* amusing<br/>
It also:<br/>
1. is not universally accepted<br/>
2. ignores historical precedent<br/>
3. encourages short paragraphs<br/>
This is still part of the second paragraph, but the above lists were not embedded correctly.</p>
<p>This is a third paragraph.</p>
```

Example 3: Correct Figure Embedding

If a figure is inserted on its own line, with single newlines above and below it, it is inserted as an inline figure inside the paragraph, at that position exactly if not floated, or near that position if floated.

Markua Syntax

```
This is paragraph 1.

Markua has a fancy logo:
  ![the word Markua with the two asterisk trees beside it](markua-logo.png "The Markua Logo")
Thanks Justin, it was fun making the logo with you!

This is paragraph 3.
```

HTML Output

TODO - do we add the br tags or not for inline figures?

```
<p>This is paragraph 1.</p>
<div class="para">Markua has a fancy logo:
  <div class="figure">
    <img src="media/markua-logo.png" alt="the word Markua with the two asterisk trees beside it"/>
    <p class="caption">The Markua Logo</p>
  </div>
  Thanks Justin, it was fun making the logo with you!
</div>
<p>This is paragraph 3.</p>
```

Syntax for index entries

```
a b c {i:C} d e

...

a b c{i:C} d e
```

results in:

```
a b <span id="__i__1__C">c</span> d e

...

a b <span id="__i__2_C">c</span> d e



The id will be `__i__<number>__<index>`. Where `index` is everything inside of the curly braces after `i:` and\
 before the `|`, with `*` replaced with nothing and `!` replaced with `__`.
index = (The thing inside the curly braces after `i:`).split(/\|/).first.gsub(/\!/, '__').gsub(/\*/, '').lower
`number` is the number of times `index` has been indexed in the document already, counting from 1.
```

Appendices

Definitions

There are a number of key terms and phrases which are used throughout the Markua specification.

book or document
: If this phrase is used to refer to the files that the author is writing in, it is used to refer to a Markua manuscript. If this phrase is used to refer to a finished product, then it is referring to the output files produced by a Markua Processor from a Markua manuscript.

ebook
: Ebooks, or electronic books, are typically available in PDF, EPUB and/or MOBI formats. A Markua manuscript can be turned into an ebook by Markua Processors that are capable of generating these formats.

Leanpub
: Leanpub is the startup that created Leanpub Flavoured Markdown and Markua. The company that created, owns and operates Leanpub is Ruboss Technology Corporation, a British Columbia, Canada company.

Leanpub Flavoured Markdown
: The format which was the predecessor to Markua. It was focused primarily on the generation of books from a superset of Markdown, minus inline HTML support.

Markua
: Markua is a plain text format designed for the writing of books and documents.

Markua manuscript
: Also called “manuscript” or “manuscript files”, a Markua manuscript is the Markua-formatted UTF-8 text file or files that the author actually writes in. The Markua manuscript is used as input by Markua Processors to generate Markua output files.

Markua output files
: The formatted output files generated from Markua manuscripts by Markua Processors. The formats generated can include ebook formats (PDF, EPUB and MOBI) as well as HTML and InDesign formats, among others. Only the mapping to HTML is precisely specified.

Markua Processor
: A Markua Processor can turn a Markua manuscript into Markua output files.

must
: This is a requirement of Markua. Markua Processors must behave this way to be compliant with this specification.

newline
: This is a carriage return, line feed, or a carriage return followed by a line feed. Pressing the enter or return key on your keyboard will produce one of these combinations on most modern operating systems.

should
: This is recommended by Markua. Markua Processors should behave this way, but do not have to.

text content
: These are letters, numbers, punctuation, etc.

whitespace
: Whitespace is one or more consecutive spaces or tabs. It does not include not newlines.

The W3C and Semantic Emphasis

Markua is a plain text format designed for the writing of books and documents.

Books and documents contain bold and italicized text–and have done so for centuries. Because of this, you'd think that the notion of making text bold or italic would be well understood.

If you think this, you are clearly not familiar with the work of the W3C. The notion of making something bold or italic is very controversial in HTML: first the `<b>` and `<i>` tags were used to do this, then they got kind of deprecated in favour of `<strong>` and `<em>`, and now there are four tags: `<b>`, `<strong>`, `<i>` and `<em>`, and *none* of these tags are guaranteed to produce something bold or italic!

So, what to do?

Markua outputs PDF, EPUB, MOBI and HTML ebooks. EPUB is essentially just zipped HTML. So, two of the for output formats of Markua are essentially HTML.

There are now four main ways of adding emphasis to text in HTML5: `<strong>`[23], `<b>`[24], `<em>`[25] and `<i>`[26]. The `<strong>` and `<b>` tags can wrap a span of text and add bold-esque emphasis to it, while the `<em>` and `<i>` tags can wrap a span of text and add italic-esque emphasis to it. I say "esque" since there is no guarantee that any of the four tags will actually make anything bold or italic. All four tags have actually become semantic tags in HTML5. Seriously.

So, what to do in Markua? Clearly we need to be able to make text bold and italic, since authors have been doing this in books for centuries. But EPUB is one of the output targets of Markua, and since EPUB is essentially just zipped up HTML, we need to use some HTML tags for empahasis.

Now, what Markdown does is to just always generate `<strong>` and `<em>` tags. Both `**two asterisks**` and `__two underscores__` make a `<strong>` tag, and `*one asterisk*` and `_one underscore_` make an `<em>` tag. This means that in 2004, John Gruber was clearly in the camp of "the `<b>` and `<i>` tags are unsemantic and bad". (At that time, many people were in this camp, as HTML had gone to a dark place full of presentational attributes in tags. The "move all presentation to CSS" backlash was very necessary, but there were a couple

[23]http://www.w3.org/TR/html5/text-level-semantics.html#the-strong-element
[24]http://www.w3.org/TR/html5/text-level-semantics.html#the-b-element
[25]http://www.w3.org/TR/html5/text-level-semantics.html#the-em-element
[26]http://www.w3.org/TR/html5/text-level-semantics.html#the-i-element

places–such as this, and such as the deprecating of the `start` element of an `<ol>`–where semantic information was lumped in with presentation-only information and was unfairly maligned.)

Now, since you can inline HTML in Markdown, if you wanted filthy `<b>` and `<i>` tags, you could just add them yourself. In Markua, however, we don't support inlining HTML, so that's not an option.

So, what to do?

Let's think about the goal of Markua:

> Markua is intended to be extremely simple and powerful. We want Markua to be the best way in the world to write everything from a short story or a novel to a computer programming book, user manual or doctoral thesis.

Given this, we want to provide the power of picking what kind of emphasis that you want, while ensuring that Markua is simple to learn. We need to support all four ways of producing emphasis in Markdown (one and two asterisks and underscores), and we want to be able to produce all four of the (newly semantic!) emphasis tags in HTML5. That's exactly what we're going to do. Unlike Markdown which defines four input tags and only two outputs, we're going to map the four inputs to four outputs.

There are actually a number of benefits to this:

- Publishers can style `<b>` and `<strong>` differently. The `<strong>` tag can produce text which is using an even heavier-weight font than the `<b>` tag.
- Authors can italicize things they are supposed to (e.g. the titles of books), without implying the kind of semantic emphasis of the `<em>` tag.
- Since the four combinations of asterisks and underscores produces different output, which to choose will be determined by the desired output, not personal taste of "do you prefer underscores or asterisks". This means that multi-author Markua projects won't have to have pointless writing style debates about which type of Markua markup to use.
- All Markdown text which has any of the four kinds of emphasis will still generate emphasis that looks like this in Markua.
- Text editors (e.g. iA Writer) which know how to show emphasis for Markdown syntax will do so with Markua emphasis syntax.

Markdown Headings

Whereas Markua defines exactly one way of making every type of heading, Markdown defines two syntaxes for making headings: "Setext" and "atx". For the purpose of contrast, this section explains how headings are done in Markdown.

Setext Headings in Markdown

In the Setext-style of headings, top level headings can be made by adding a row of equals signs ======== below the heading, and second level headings can be made by adding a row of minus signs -------- below the heading.

These are some Setext headers in Markdown:

```
This is an h1
=============

This is a paragraph.

This is also an h1 (really)
=

This is a paragraph.

This is an h2
-------------

This is a paragraph.

This is also an h2 (really)
-

This is a paragraph.
```

There are two main issues with this style of heading in Markdown:

1. It is inconsistent: It only supports defining `h1` and `h2`; to define h3 and below, the atx style of heading must be used.
2. It is confusing: Authors are confused about how many equals or minus signs you need to use. (As we saw above, the answer is 1, which looks disgusting.) Also, it's unclear whether you need to add a blank line below the row of equals signs or minus signs, and whether a heading is still produced if this is not done.

atx Headings in Markdown

In Markdown, the other heading syntax is "atx" headers. The atx format[27] was created by Aaron Swartz in 2002, and is an important predecessor to Markdown. To create an atx heading, you put the following on a line by itself, with a blank line above and below it: between one and six pound signs (`#`), followed by one or more spaces or tabs, followed by text, followed optionally by whitespace and any number of `#` characters. The number of `#` characters determines the level of heading.

These are some atx headers in Markdown:

[27]http://www.aaronsw.com/2002/atx/intro

```
# This is an h1

This is a paragraph.

# This is also an h1 ####################

This is a paragraph.

## This is an h2

This is a paragraph.

## This is also an h2 ###################

This is a paragraph.

### This is an h3

This is a paragraph.

### This is also an h3 ##################

This is a paragraph.
```

Note that all the trailing `#` characters are ignored and are not part of the heading text.

From reading this, you'll recogrize that Markua headings are a subset of atx headers.

⚠ TODO_LEANPUB - bug - h6 needs proper newlines afterward, and needs to be numbered if numbering on

Important Changes From Leanpub Flavoured Markdown

Markua is the evolution of Leanpub Flavoured Markdown. However, the goal of Markua is to remove the tailbones and appendices that aren't necessary. This appendix highlights things which are changed from Leanpub Flavoured Markdown in a way which is not adequately discussed earlier.

The following Leanpub Flavoured Markdown features are not supported in Markua:

- Centering paragraphs of text or headings using `C>` has been removed.
- Explicit font switching via `{chinesefont}`, `{latinfont}`, `{japanesefont}`, `{koreanfont}` and `{thaifont}` is replaced with attaching a `{lang}` attribute (e.g. `[some text]{lang:en}`) on a span or block.
- Various Leanpub Flavoured Markdown directives which were added willy-nilly over the years are being dropped, including `{ltr}`, `{rtl}`, `{begin-hanging-paragraphs}`, `{end-hanging-paragraphs}`, `{footnotes-on}`, `{footnotes-off}`

Rules and Guidelines for Implementors of Markua Processors

This entire spec is a set of rules and guidelines, but there are a few specific things to call out here:

- Markua Processors should have one comprehensive list of warnings to show Markua authors upon generating a Markua document with warnings in it.
- Markua Processors must not output generation errors and warnings into the text itself.
- Markua processors should provide warnings in situations where they can detect issues (e.g. "duplicate cross-reference", "not understood programming language", "unknown math processor", etc).
- Markua processors must not explode when encountering unsupported features that have legal Markua syntax (e.g. an unknown Math dialect or human language).
- Links to cross-references that don't correspond to an id that exists may either be created as a (broken, non-functional) link or be created as normal text (not a link).
- Crosslinks that reference an unused id may either be created as a (non-functional) link or be created as normal text (not a link) by a Markua Processor. The Markua Processor may also output a warning about this somewhere, but not in the actual document text itself.

Required and Optional Resource Support

Current copyright law makes the use of resources problematic. As such:

- Support for `image`, `audio` and `video` resources is optional in Markua processors, regardless of location.
- Support for web resources is optional in Markua processors, regardless of resource type.
- Support for web resources is not all or nothing–a Markua Processor can support web resources for some resource types but not others.

For example, a Markua Processor could support web resources for code, math and text, but not for images, audio and video. If a Markua Processor does not support a particular type of web resource, it must consider all web resources of to be missing resources at the time of book or document generation, and use fallback resources or alt text if provided.

Acknowledgments

I (Peter Armstrong[28]) am the creator of Markua, but I'm getting lots of help.

First, a huge thanks to Scott Patten[29], my co-founder at Ruboss and Leanpub. Scott is the lead developer of Leanpub's book generation engine. Leanpub Flavoured Markdown would not have existed without Scott's code. Furthermore, Scott is providing valuable feedback in the development of the Markua specification, as well as adding Markua support to Leanpub's book generation engine. Scott is giving me opinionated feedback on the Markua syntax, just as Aaron Swartz gave John Gruber feedback on the Markdown syntax. In many instances, e.g. the poetry syntax, what happens is that I come up with an idea which is good but flawed, Scott points the flaws out and suggests an alternative which often feels like giving into the status quo or feels too much like programming, I get really mad, and then I come up with something better than I would have otherwise. So, Markua probably also wouldn't exist if I couldn't–repeatedly and productively–get mad at Scott.

Second, I'd like to thank Braden Simpson[30]. Braden has been instrumental in the development of the open source ES6 Markua parser, as well as the open source MarkuaPad editor and its usage in Leanpub. Braden has also been key in developing the test suite for Markua, and along with Scott has given me great feedback on crucial design decisions such as the image syntax.

Third, an ongoing thanks to Len Epp[31] and Jordan Ell[32], who have been instrumental in the development of Leanpub. Len has been with us since the very early days, and has greatly helped our customer development process, helping (as he likes to tease me) grow Leanpub over 100x since joining us. Jordan is proving key in helping Scott, Braden, Len and me to bring Leanpub–and Markua–to the next order of magnitude of success.

I'm getting great feedback on Twitter as well. Of particular note are the contributions of Iva Cheung[33], Assaf Arkin[34], Axel Rauschmayer[35] and Michael Müller[36]. Iva helped us figure out how to think about front matter, main matter and back matter properly, and she also organizes a Vancouver publishing unconference that I've had the pleasure of both speaking at and of sponsoring. I know Assaf from my Silicon Valley days: Assaf was the CTO of Intalio for the four years that I was a developer there. Assaf is probably the smartest technologist that I have ever worked with (sorry Ken!), and his interest in Markua and his feedback during its development were both motivating and helpful. Axel was really helpful to me in an email thread that we had, which was the catalyst for me to rethink my goals with Markua and to start throwing out more than I had planned to from Leanpub Flavoured Markdown. Michael was key in convincing me to add index entry support to Markua.

Perhaps most important, Markua is benefiting from years of feedback about Leanpub Flavoured Markdown from Leanpub authors. If it wasn't for Leanpub authors, there would be no Markua. Some Leanpub authors are providing helpful feedback over email. Thanks especially to Dave Warnock.

[28]https://twitter.com/peterarmstrong
[29]https://twitter.com/scott_patten
[30]https://twitter.com/bradensimpson
[31]https://twitter.com/lenepp
[32]https://twitter.com/jordanelltweets
[33]https://twitter.com/IvaCheung
[34]https://twitter.com/assaf
[35]https://twitter.com/rauschma
[36]https://twitter.com/muellermi

I'd also like to thank Justin Damer[37]. Justin and I designed the logo for Markua together, just as we designed the Leanpub logo together. I've never enjoyed working with a designer more than I enjoy working with Justin.

Next, a huge thanks to Ken Pratt[38], who made a number of key contributions to the development of Leanpub Flavoured Markdown and to Ruboss over his years working with us.

Finally, I would like to thank my wife Caroline and my son Evan: while this is not as long as some of my books, there's still a lot of thought and effort going into it, and a lot of sacrifice. Arguably, this is harder than any book I've ever written. As always, their support is very much appreciated.

[37]https://twitter.com/damerdamer
[38]https://twitter.com/ken_pratt