Markless Specification

v0.8-2-gaa76075

Contents

1 Preamble

Markless is a new markup standard that focuses on being intuitive and fast to parse. Being a purely text-based markup, no complicated editor software is required to create documents in it. With its focus on intuition and consistency it should also be a good fit as a markup choice for text based platforms such as chat, forums, etc. Markless does not specify its results based on another document format, meaning that an implementation could be written to turn a Markless document into practically any other format. Markless is strict and does not allow for any ambiguities in its markup. This both makes it less confusing for the user, and easier to parse for a program. Being based on a specification rather than a reference implementation, Markless also offers the users a much more stable and reliant source to turn to in case of questions about the behaviour of an implementation.

This document specifies the way a Markless *document* is treated and how the various markup *directives* are to be *interpreted*. It does not describe the technological aspects of writing an *implementation* for Markless. It should also not be used as a guide or introduction on how to write Markless documents, but rather as a reference if you should want to write a new *implementation* or are unsure about the behaviour of an existing one. This document also describes the terminology to allow talking about Markless terms unambiguously. See the *glossary* for reference.

Included in most sections are one or more examples. These examples exist purely for illustrative purposes and are not normative. An *implementation* may deviate from the behaviour illustrated by the examples as long as it adheres to the actual description of this specification.

2 Identifier Syntax

In order to concisely specify *identifiers* we use a special syntax, of which the full grammar and semantics are reflected here using BNF notation.

`rule`	::=	"("? (`matcher quantifier`?)+ ")"?
		`rule` | `string` | `some-characters`
`matcher`	::=	| `any-character` | `not` | `either` | `binding`
		| `binding-reference` | `identifier-reference`
`string`	::=	`character`+
`char-class`	::=	"~" `character`
`some-characters`	::=	"[" `character`+ "]"
`any-character`	::=	"."
`not`	::=	"!" `matcher`
`either`	::=	`rule` "|" `rule`
`binding`	::=	"<" `name` " " `rule` ">"
`binding-reference`	::=	"<" `name` ">"
`identifier-reference`	::=	"{" `name` "}"
`quantifier`	::=	`one-or-more` | `none-or-more` | `one-or-none`
`one-or-more`	::=	`rule` "+"
`none-or-more`	::=	`rule` "*"
`one-or-none`	::=	`rule` "?"
`name`	—	Some *alphanumeric string* to identify the text matched by the `rule`.
`character`	—	A *character*.

Appearing within the "" quotes are *characters* to be found in the *identifier specifier*.

If a backslash appears anywhere within the *identifier specifier*, it is ignored and the *character* immediately after it is taken literally without being interpreted as one of the *characters* in the syntax rules and without being interpreted using this backslash rule. Thus two backslashes immediately after one another are interpreted as a single, literal backslash *character*.

In order for a `rule` to *match*, the `quantifier` supplied with the `matcher` must match. If no `quantifier` is included in a `rule`, the `rule` *matches* if the `matcher` *matches* exactly once.

In order for a `string` to *match*, the exact sequence of *characters* must be found.

In order for a `char-class` to *match*, a *character* specified by the *character class* associated with the given `character` must be found. The following classes are specified: `a` for *alphabetic*, `n` for *numeric*, `_` for *whitespace*, and `w` for *alphanumeric*.

In order for `some-characters` to *match*, one of the *characters* must be found.

In order for `any-character` to *match*, a single *character* must be found, but it matters not which *character* it is.

In order for `not` to *match*, the following `matcher` must not *match*.

In order for `either` to *match*, either the `rule` left to it, or the `rule` right to it must *match*.

In order for `one-or-more` to *match*, the `rule` must be *matched* at least once, but may be *matched* an

arbitrary number of times immediately after each other. The `rule` is only repeatedly *matched* until the `rule` immediately after the `one-or-more` is *matched*.

In order for `none-or-more` to *match*, the `rule` does not have to be *matched* at all, but may be *matched* an arbitrary number of times immediately after each other. The `rule` is only repeatedly *matched* until the `rule` immediately after the `none-or-more` is *matched*.

In order for `one-or-none` to *match*, the `rule` does not have to be *matched* at all, but if it is, it is only *matched* exactly once.

In order for a `binding` to *match*, the `rule` contained must *match*. The specific *string matched* by the `rule` is then associated with the `name` of the `binding`.

In order for an `identifier-reference` to *match*, the *identifier* corresponding to the `name` must *match*. The effect is as if the according *identifier specifier* was used in place of the `identifier-reference`.

In order for a `binding-reference` to *match*, the exact *string* associated with the `name` of the `binding` must be found.

3   Documents

Markless describes a number of *directives* to transform a *document* from its bare *string* representation into that of a *textual component*. While the *directives* are described in this specification using Unicode *characters*, the specification does not enforce any particular *encoding* on the *document*. However, in order for an *implementation* to be *conforming*, *characters* used to identify a *directive* in a *document* must be *equivalent* to those in this specification.

The effect of a *textual component* on its *text* applies on all *levels*. In the case of conflicting *styles*, the *style* of the *textual component* on the closest *level* above the *text* applies. In effect this means that a *textual component* on a lower *level* can override a *style* for its *text*.

An *implementation* may choose to compose multiple *textual components* in order to achieve the effect of a single *specified textual component*. It may also insert *textual components* at any point in the *document* if necessary by the resulting *document format*. An *implementation* may also ignore any *style* of a *specified textual component* if the resulting *document format* cannot support its effect.

4   Interpretation

This section describes the procedure by which an *implementation interprets* a *document*. This procedure is used as a reference to allow verification of correctness. An *implementation* does not necessarily have to follow this procedure as long as the output it produces is equivalent with an *implementation* that does.

4.1   State

The following state is kept and updated as the procedure advances.

- The input stream from which characters are read.

- The parser state variables such as the *line break mode*.

- A *cursor*.

- A stack wherein each entry is composed of a *directive* and a *textual component*.

- A list of *disabled directives*.

- A table associating *labels* to *textual components*.

4.2   Procedure

1. A "root-directive" and a "root-component" are pushed onto the stack.

2. If the input stream has things to read:

    2.1. A *line* is read from the input stream.

    2.2. The *cursor* is set to the beginning of the *line*.

    2.3. The stack is traversed upwards from the bottom:

        2.3.1. The *directive* at the current stack entry attempts to *match*.

        2.3.2. If the *match* succeeds:

            2.3.2.1. The *cursor* is advanced by the *matched characters*.

            2.3.2.2. The current stack entry is advanced upwards.

            2.3.2.3. Go to 2.3.1.

        2.3.3. The stack is unwound down to and including the current stack entry. See *Stack Unwinding*.

    2.4. The *directive* on top of the stack is invoked:

        2.4.1. If an *applicable directive matches*:

            2.4.1.1. The *matched directive* may enter *textual components* into the *current component*.

            2.4.1.2. The *matched directive* may push itself and a *textual component* onto the stack or perform other changes to the state as specified.

            2.4.1.3. The *cursor* is advanced by the *matched characters*.

            2.4.1.4. Go to 2.4.

        2.4.2. The *character* at the *cursor* is added to the *current component*.

        2.4.3. The *cursor* is advanced by the *character*.

    2.5. If the *cursor* is not yet at the end of the *line*:

        2.5.1. Go to 2.4.

    2.6. If the *line break mode* is `show`, a *newline* is added to the *current component*.

2.7. Go to 2.

3. The stack is unwound fully. See *Stack Unwinding*.

4. The interpretation is complete. The "root-component" represents the resulting *document*.

4.2.1 Stack Unwinding

1. If the stack is taller than the desired height:

    1.1. If the directive on top of the stack is a *inline directive*:

        1.1.1. The *current component* is converted to one that has no *style*.
        1.1.2. *Characters* that have been consumed by the prefix match of the *directive* are prepended to the *current component*.
        1.1.3. Other potentially necessary actions to undo the match of the *directive* are performed.

    1.2. The top of the stack is popped off.

    1.3. Go to 1.

4.2.2 Root Directive

The root directive always *matches* but is never considered an *applicable directive*.

5 Line Directives

In order for a *directive* to be a *line directive*, its *identifier* must *match* the beginning of a *line*.

A *textual component* specified by a *line directive* can potentially contain any other *textual component*. Therefore, any *directive* is potentially recognisable within a *line directive*, including other *line directives*. However, a *line directive* may explicitly restrict which *directives* are recognised within itself. A *line directive* cannot cross the boundaries of another *line directive* of a different kind. If such a case were to occur, the current *line directive* is forcibly ended without regard for any possible trailing *match*.

5.0.1 Singular Line Directives

A *line directive* is a *singular line directive* if it is only ever active for a single *line*. If it is matched on two consecutive *lines* this results in two separate *resulting textual components*.

When a *singular line directive* is *processed*, processing begins anew over the *content binding* until the end of the *line* is reached, at which point the *resulting textual component* is ended. After that, control is handed back to the *standard processing loop*.

5.0.2 Spanning Line Directives

A *line directive* is a *spanning line directive* if the *identifier* contains a *content binding*, and if *matches* on consecutive *lines* of the *identifier* are interpreted as a single *match*. The semantics of such a spanning match are as follows: Only a single *resulting textual component* is produced for all the consecutively *matching lines*. The *text* of this *resulting textual component* is produced by concatenating the contents of the *content binding* on each *line*. If the *content binding* does not *match* the *newline* on every *line*, the *newline* must be inserted between each *string* of the *content binding*.

When a *spanning line directive* is *processed*, processing begins anew over the *content binding* until the end of the *line* is reached. Standard end of *line interpretation* proceeds. If the following *line matches* the same *spanning line directive* as before, processing begins anew over the *content binding* thereof without any new *resulting textual components* being started or inserted. If the following *line* does not *match* the same *spanning line directive* as before, the *resulting textual component* is ended and control is handed back to the *standard processing loop*.

5.0.3 Guarded Line Directives

A *line directive* is a *guarded line directive* if its *matched* region is specified by two *identifiers* that each match a single *line*. The *text* of the *resulting textual component* is the *text* from the *line* immediately after the *line* the first *identifier matches* until and including the *line* immediately before the *line* the second *identifier matches*.

When a *guarded line directive* is *processed*, processing begins anew over the *content binding* until the the part of the *identifier* after the *content binding* is *fully matched*, at which point the *resulting textual component* is ended and control is handed back to the *standard processing loop*.

5.1 Paragraph

Identifier **Paragraph**:

```
<spaces [ ]*><content![ ].*>
```

Textual Component **Paragraph**: `margin: top, bottom`

The paragraph can only be *matched* if no other *line directive matches.* *Lines* belong to the same paragraph until the length of `spaces` changes, a new *inline directive* is recognised, or an *empty line* is encountered. The paragraph is a *spanning line directive.*

Paragraphs are visually distinguished by a margin above and below the *text.* An *implementation* may additionally employ indentation rules to distinguish the beginning of a paragraph.

Examples:

```
This is a paragraph
that spans multiple lines
```
⇒

This is a paragraph

that spans multiple lines.

```
This is another paragraph.
```

This is another paragraph.

```
Paragraph One
  Paragraph Two
```
⇒

Paragraph One

Paragraph Two

5.2 Blockquote

Identifier **Blockquote Header**:

`\~ <content !(\| )+>(\| <body .*>)?`

Identifier **Blockquote Body**:

`\| <content .*>`

Textual Component **Blockquote Header**: `margin: left; font-weight: bold`
Textual Component **Blockquote Body**: `margin: left`

The blockquote header is a *singular line directive* that identifies the source of a quote. Only the *text* held by the *content binding* is outputted into the *resulting textual component.* The blockquote header *content binding* can only contain *inline directives.* If the `body` *binding* is present, then the blockquote header's *resulting textual component* is closed, and parsing resumes as if at a block toplevel, meaning a blockquote body will be parsed next. This allows a shorter form of combined header and body.

The blockquote body is a *spanning line directive* that identifies a body of *text* that is being quoted. The blockquote body can contain any *directive* with the condition that the *directives* are matched against the *text* of the *resulting textual component.*

An implementation may choose to group the *blockquote header* and *blockquote body* together and reorder them if they are found consecutive to one another. However, a body can only ever be grouped together with a single header. In the case where a header lies between two bodies, the header is counted to belong to the second body. If a header is found without a corresponding body, the *implementation* may *signal* a *warning.*

Examples:

```
~ This Document
| The blockquote header is a \
| singular line directive.
```
⇒

The blockquote header is a singular line directive.

— This Document

```
| Unattributed text.
```
⇒
Unattributed text.

```
~ Yukari | Hello there!
```
⇒
Hello there!

— Yukari

5.3 Lists

Identifier **Ordered List**:

```
<number ~d+>\.<content .*>
(<spacing ~_+> <content .*>)*
```

Identifier **Unordered List**:

```
- <content .*>
(<spacing ~_+> <content .*>)*
```

Textual Component **Ordered List**: `margin: left`
Textual Component **Ordered List Item**: `display: list-item; list-item-prefix: number`
Textual Component **Unordered List**: `margin: left`
Textual Component **Unordered List Item**: `display: list-item; list-item-prefix: dot`

The lists are *spanning line directives* and mark the enumeration of one or more items of a list. They can contain any *directive* with the condition that the *directives* are matched against the *text* of the *resulting textual component.*

After the respective list *identifier* has been *matched*, a new respective item *textual component* in which the higher *level text* is contained, is inserted for each *match* into the spanning *resulting textual component*. A single *match* may span over multiple *lines* if the *text matched* by the `spacing` *binding* is of the same length as that of the `number` *binding*. In such a case, each item *match* itself is treated like a *spanning line directive* where the *content binding* is concatenated.

Ordered list items must be numbered by the *decimal number* given by the `number` *binding*, even if there is no order to how the numbers appear in the list or if there are duplicates.

Examples:

```
- Finish this spec
- Implement a parser
```
⇒
- Finish this spec
- Implement a parser

```
1.Buy some ingredients
2.Clean the kitchen
  Don't forget the sink!
5.Watch TV
```
⇒
1. Buy some ingredients
2. Clean the kitchen
   Don't forget the sink!
5. Watch TV

5.4 Header

Identifier **Header**:

```
<level #+> <content .+>
```

Textual Component **Header**: `font-weight:bold; font-size: 1-level; indent: true; label: content`

The header is a *singular line directive.* It represents a section heading. Only the *text* held by the *content binding* is outputted to the *resulting textual component.* The header can only contain *inline directives.*

The length of the `level` *binding* determines the level of the heading. The level may potentially be infinitely high, though the *implementation* may represent levels above a certain number in the same manner. It must however support a different representation for at least levels 1 and 2. Generally, the higher the level, the smaller the font size of the heading should be.

An *implementation* may choose to number each header, where this number prefix is put together by the number prefix of the header on a level one higher followed by a dot and a counter representing how many headers of the same level have appeared until and including the current one since the last header of a higher level. In the case of a level one heading only the counter is used, as there is no higher level prefix to prepend. In the case where no level one higher is contained in the *document*, the level is treated as if it existed with the counter for it being 0.

The *resulting textual component* is associated with a *label* of the same name as the *text* of the *resulting textual component.*

Examples:

```
# Header
The header is a singular line
directive
## Subsection
That allows neat sectioning!
```

$\Rightarrow$

Header

The header is a singular line directive.

Subsection

That allows neat sectioning!

```
# Cooking a Lasagna
Here's what you have to buy:
## Ingredients
A buncha stuff!
## Steps
It's a lengthy recipe, but finally \
you'll have to
#### Bake it
```

$\Rightarrow$

1 Cooking a Lasagna

Here's what you have to buy:

1.1 Ingredients

A buncha stuff!

1.2 Steps

It's a lengthy recipe, but finally you'll have to

1.2.0.1 Bake it

5.5 Horizontal Rule

Identifier **Horizontal-rule**:

`==+`

Textual Component **Horizontal-rule**: `display: line`

The horizontal rule is a *singular line directive.* It is translated into a *resulting textual component* that represents a horizontal rule or break on the page. This must span the entire width of the document and could be represented by a thin line. If the *document* cannot support the drawing of lines, the horizontal rule may instead be approximated through other means.

Examples:

```
==
```
⇒ _____

```
And now, for a brief break.
```
And now, for a brief break.
```
=====
```
⇒ _____

```
Back to the show!
```
Back to the show!

5.6 Code block

Identifier **Code Block**:

```
<prefix ::+> *<language ![,]+>?<options .*>
<content .*>
<prefix>
```

Textual Component **Code Block**: font-family: monospace; white-space: preserve

The code block is a *guarded line directive*. It marks the *text* to belong to a *textual component* that somehow distinguishes the block as source code. Only the *text* held by the *content binding* is outputted to the *resulting textual component*. The code block *directive* cannot contain any other *directives*.

The *newlines* and *whitespace* must be represented exactly as in the source text. Multiple consecutive *whitespace characters* cannot be combined and must be individually represented. A *newline character* cannot be escaped and must always result in a new line being started. *Escaping* is deactivated within the content, meaning backslashes are output literally in the *resulting textual component*.

The `options` *binding* holds potential parameters that can configure the *style* of the *resulting textual component*. The syntax and effect of the options is *implementation dependant*. If a language is requested in the *language binding* that the *implementation* does not have specific support for, a *warning* is *signalled*. The language `text` must always be supported and will incur no styling changes.

Examples:

```
Some unexciting code:
:: common-lisp
(print "Hello world")
::
```
⇒
Some unexciting code:

(**print** "Hello world")

5.7 Instruction

Identifier **Instruction**:

```
\! <instruction .*>
```

The instruction is a *singular line directive*. Its purpose is to interact with the *implementation* and cause it to perform differently. There is no corresponding *resulting textual component* for the instruction *directive*.

An *implementation* is allowed to add further instructions. If an instruction is not recognised, the *implementation* must *signal* an *error*.

5.7.1 Set

Instruction **Set**: set <variable ![ ]+> <value .+>

Sets the state of the *variable* of the given name to a certain value. An *implementation* may check the value for validity and *signal* an *error* if it is invalid. An *implementation* is allowed to add further variables. If a variable is not recognised, the *implementation* must *signal* an *error*.

5.7.1.1 Line Break Mode
Variable **line-break-mode**: `show`

The *line-break-mode* variable may only assume two values: `show`, and `hide`. If the line break mode is `show`, when the processor encounters an unescaped *newline*, a new *line* is started in the output *document*.

Examples:

```
! set line-break-mode show
foo
bar\
baz
! set line-break-mode hide
bada
boom
```

$\Rightarrow$

foo

barbaz

badaboom

5.7.1.2 Metadata
Variable **author**:
Variable **copyright**:
Variable **language**:

Declares *metadata* about the *document*. The *implementation* may use this information and embed it into the output *document*.

5.7.2 Message

Instruction **Info**: `info <message .*>`
Instruction **Warn**: `warn <message .*>`
Instruction **Error**: `error <message .*>`

The *info* instruction causes the *implementation* to *signal* the given message. The *warn* instruction causes the *implementation* to *signal* a *warning* with the given message. The *error* instruction causes the *implementation* to *signal* an *error* with the given message.

5.7.3 Include

Instruction **Include**: `include <file .*>`

Causes the implementation to *interpret* the contents of the given file. If the file is not accessible for some reason, the *implementation* must *signal* an *error*.

5.7.4 Directives

Instruction **Disable**: `disable <directive ![ ]+>( <directive ![ ]+>)*`
Instruction **Enable**: `enable <directive ![ ]+>( <directive ![ ]+>)*`

The *disable* and *enable* instructions cause the *implementation* to respectively disable or enable the named *directives.* If a given name is not recognised, the *implementation* may *signal* a *warning.*

Examples:

```
! disable instruction
! error Exit!
```
⇒ ! error Exit!

5.7.5 Label

Instruction **Label**: `label <name .*>`

Associates a *label* with the component that immediately precedes this instruction.

5.7.6 Raw

Instruction **Raw**: `raw <backend ![ ]+> <content .*>`

If the *implementation*'s chosen output backend matches that of the `backend` *binding*, the *implementation* should emit the `content` *binding*'s text verbatim into the resulting document. This should allow creating output specific effects. The exact semantics and results of this are *implementation dependant.* If the *implementation*'s chosen output backend does not match, the instruction is ignored.

Users should note that basic Markless parsing rules such as backslash escapes still apply for the `content`, so the content is not copied directly 1:1 from the source text to the output document.

Examples:

```
! raw latex \\textit{Hello}
```
⇒ *Hello*

```
! raw html <blink>Hello</blink>
```
⇒

5.8 Comment

Identifier **Comment**:

```
;+ .*
```

The comment is a *singular line directive.* If the *comment identifier* is *matched*, the entire line is skipped and discarded. There is no corresponding *resulting textual component* for the comment *directive* and as such it must not have any effect on the *document.*

Examples:

```
; This is a stupid thing to say.
Sometimes
;forever
```
⇒
Sometimes

;forever

5.9 Embed

Identifier **Embed**:

```
\[ <type .*> <target ![,]*>(, *<parameter ![,]*>)*( *\])?
```

Textual Component **Embed**: `display: block;target: target`

The embed is a *singular line directive.* The content of the `type` binding determines the embed's type, and the `parameter` bindings determine the embed's parameters. The style of the *resulting textual component* is dynamically dependant on the given type.

Unless the *embed-property-width* or *embed-property-height* parameters are present, the size of the embed *resulting textual component* is constrained to be smaller than the width and height of the *page* it is output to while preserving the embed content's aspect ratio. If the *page* has no width or height, or the embed content's dimensions are smaller than both of those, then the embed content is sized to its own dimensions. The *resulting textual component* must not be split across multiple *pages.*

5.9.1 Embed Types

An *implementation* must at least support the types specified in this section if permitted by the output *document*, but may add additional options the implications of which are completely *implementation dependant.* If the output *document* does not support a particular type, a *paragraph* containing a single *url textual component* is outputted with its target set to the `target` *binding*'s value, its content set to either the content of the `description` option if present, or the `target` *binding*'s value if not, and a *warning* is *signalled.* If the *implementation* does not support the requested type at all, an *error* is *signalled.*

5.9.1.1 Image
Identifier **Embed-type-image**: `image`
Style **Embed-type-image**: `interaction: image`

Embeds the image pointed to by the `target` into the document. The supported image formats are *implementation dependant.* If the format of the target is not supported by the *implementation*, the *directive* is treated as if it were given an unknown type.

Examples:

`[ image markless-logo.png ]` ⇒

5.9.1.2 Video
Identifier **Embed-type-video**: `video`
Identifier **Embed-property-loop**: `loop`
Identifier **Embed-property-autoplay**: `autoplay`
Style **Embed-type-video**: `interaction: video`

Embeds the video pointed to by the `target` into the document. The supported video formats are *implementation dependant.* If the format of the target is not supported by the *implementation*, the *directive* is treated as if it were given an unknown type. The *resulting textual component* must be interactive in such a way that the *user* is presented with a way to start, pause, seek, and change the volume of the video. The video should not play automatically, unless the *embed-property-autoplay* flag property is present. If the *embed-property-loop* flag property is present, the video should start over from the beginning once it

reaches the end.

Examples:
```
[ video sample.mp4 ]  ⇒   file://./sample.mp4
```

5.9.1.3 Audio
Identifier **Embed-type-audio**: `audio`
Identifier **Embed-property-loop**: `loop`
Identifier **Embed-property-autoplay**: `autoplay`
Style **Embed-type-audio**: `interaction: audio`

Embeds the audio file pointed to by the `target` into the document. The supported audio formats are *implementation dependant*. If the format of the target is not supported by the *implementation*, the *directive* is treated as if it were given an unknown type. The *resulting textual component* must be interactive in such a way that the *user* is presented with a way to start, pause, seek, and change the volume of the audio. The audio track should not play automatically, unless the *embed-property-autoplay* flag property is present. If the *embed-property-loop* flag property is present, the audio track should start over from the beginning once it reaches the end. Since an audio file does not have any dimensions associated with it, the *implementation* is free to choose the sizing it deems appropriate.

Examples:
```
[ audio sample.mp3 ]  ⇒   file://./sample.mp3
```

5.9.1.4 Source
Identifier **Embed-type-source**: `source`
Identifier **Embed-property-options**: `options <options .+>`
Identifier **Embed-property-language**: `language <language .+>`
Identifier **Embed-property-start**: `start <start ~n+>`
Identifier **Embed-property-end**: `end <end [+]?~n+>`
Identifier **Embed-property-encoding**: `encoding <encoding .+>`
Style **Embed-type-source**: `font-family: monospace; white-space: preserve`

Embeds the source code pointed to by the `target` into the document. To do this, the file is read as a text file in the *encoding* specified by *embed-property-encoding*. If the *embed-property-encoding* is not given, UTF-8 *encoding* is assumed. If an encoding is requested that the *implementation* does not support, an *error* is *signalled*.

The file's contents are split into a sequence of lines. If *embed-property-start* is given, as many lines as indicated in its `start` *binding* are discarded from the front. If *embed-property-end* is given, and its `end` *binding* starts with a + (`U+2B`), as many lines as indicated in the *binding* are output into the *resulting textual component*. If *embed-property-end* is given, but its *binding* does not start with + (`U+2B`), the lines until and including the line indicated by the `end` *binding* are output into the *resulting textual component*, counting the first line read from the file as the line numbered `1`.

A line in this context is determined as follows: each line in the file is delimited by either the beginning of the file, the end of the file, or the nearest Linefeed (`U+A`) characters. This means that unlike a Markless *line*, the Linefeed (`U+A`) end of line marker cannot be escaped.

The *embed-property-language* and *embed-property-options* options hold parameters that configure the *style* of the *resulting textual component*. The syntax and effect of the options is *implementation dependant*,

but it must be the same as for the *code block directive*. If a language is requested that the *implementation* does not have specific support for, a *warning* is *signalled*.

Examples:

```
[ source 5.9.1.2-source.tex, language tex, end 2 ]    ⇒
```

```
\definesubsubsection{Source}
\defineidentifier{embed-type-source}{source}
```

5.9.2   Embed Parameters

The parameters are processed in the order they are given and can effect both the content of the *resulting textual component* as well as its *style*. A parameter may also affect the processing of parameters after it. Two general types of parameters are defined: flag parameters and value parameters. Flag parameters are single parameters that add or remove an attribute from the *resulting textual component*'s *style*. Value parameters add an attribute whose value is determined by the parameter following the current one. The following parameter is then skipped over and thus not processed.

An *implementation* must at least support the parameters specified in this section if permitted by the output *document*, but may add additional parameters the implications of which are completely *implementation dependant*. If the output *document* does not support a particular parameter, or an unknown parameter is given, a *warning* is *signalled*.

5.9.2.1   Float
Identifier **Embed-property-float**: `float <orientation left|right>`
Style **Embed-property-float**: `float: orientation`

Causes the embed to float on either the left or right side of the *document*. All the *resulting textual components* after it will flow around it.

5.9.2.2   Width
Identifier **Embed-property-width**: `width (<pixels ~n+px>|<percent ~n+%>)`
Style **Embed-property-width**: `width: size`

Causes the embed content's width to be fixed to the specified size. The size can be given in either `pixels` or `percent` where `pixels` will set the width to be the exact amount of pixels given if the document is viewed at its native resolution. `percent` will scale the width to the given percentage of the width of the *document*. If the *document* should not have a width, the `percent` specification does nothing. Unless the *embed-property-height* is also specified, the embed content's aspect ratio must be preserved.

Examples:

```
[ image markless-logo.png, width 50px ]   ⇒      ! MARKLESS
```

5.9.2.3   Height
Identifier **Embed-property-height**: `height (<pixels ~n+px>|<percent ~n+%>)`
Style **Embed-property-height**: `height: size`

Causes the embed content's height to be fixed to the specified size. The size can be given in either `pixels` or `percent` where `pixels` will set the height to be the exact amount of pixels given if the document

is viewed at its native resolution. `percent` will scale the height to the given percentage of the height of the *document*. If the *document* should not have a height, the `percent` specification does nothing. Unless the *embed-property-width* is also specified, the embed content's aspect ratio must be preserved.

Examples:

```
[ image markless-logo.png, width 50px, height 100px ]  ⇒
```


5.9.2.4   Label
Identifier **Embed-property-label**: `label <name .+>`

Causes the embed to be associated with a *label* of the given `name`.

5.9.2.5   Caption
Identifier **Embed-property-caption**: `caption <content .*>`
Style **Embed-property-caption**: `interaction: image`

Causes a *textual component* to be output either alongside or within the embed's *textual component*. The text may contain any *inline directives*. The *text* held by the *content binding* is outputted to this additional *resulting textual component*. The text can only contain *inline directives*.

Examples:
```
[ image markless-logo.png, caption The //Markless// logo image. ]
```

⇒

Figure 1: The *Markless* logo image.

5.9.2.6   Description
Identifier **Embed-property-description**: `description <content .*>`

A textual description of the embed's content for use when the embed content cannot be displayed, or when the *user* employs a reader or other form of aid system that relies entirely on textual representation.

5.10   Footnote

Identifier **Footnote**:

`\[<number ~n+>\] <content .+>`

Textual Component **Footnote**:

The footnote is a *singular line directive*. Outputted to the *resulting textual component* is the *text* held by the `number` *binding* followed by a : (U+3A), followed by the *text* held by the *content binding*. The

footnote can only contain *inline directives.*

Unlike other *directives* the footnote's *resulting textual component* cannot be placed where the *identifier* is found. It must be placed such that it is at the end of a *page* in the *document.*

The *resulting textual component* is associated with a *label* with the name being the content of the `number` *binding.*

Examples:

```
Examples[1] are not authoritative.

[1] Examples are things like this.
```

$\Rightarrow$

Examples[1] are not authoritative.

─────────────────

1: Examples are things like this.

5.11   Alignment

Identifier **Left Align**:

```
\|\<<content .*>
```

Identifier **Right Align**:

```
\|\><content .*>
```

Identifier **Center**:

```
\>\<<content .*>
```

Identifier **Justify**:

```
\|\|<content .*>
```

Textual Component **Left Align**: `text-align: left`
Textual Component **Right Align**: `text-align: right`
Textual Component **Center**: `text-align: center`
Textual Component **Justify**: `text-align: justify`

All alignment directives are *spanning line directive* that change the alignment of a body of *text*. The alignment content can contain any *directive* with the condition that the *directives* are matched against the *text* of the *resulting textual component.*

Examples:

```
|< Left
>< Center
|> Right
```

$\Rightarrow$

Left

Center

Right

6 Inline Directives

A *directive* is an *inline directive* if its identification is not bound to *lines*. Unlike *line directives* therefore it can potentially be identified at any point in a string and span any length.

Any *textual component* specified by an *inline directive* can only contain *textual components* specified by *inline directives*. Furthermore, an *inline directive* cannot contain another *inline directive* of its own type at any *level*. An *inline directive* may further restrict which *directives* may appear within itself. An *inline directive* cannot cross the boundaries of another *directive* of a different kind. If such a case were to occur, the current *inline directive* is forcibly ended without regard for any possible trailing *match*. A special exception is made in the case of *spanning line directives*: since a *spanning line directive* is the combination of multiple matches of the same kind on consecutive lines into a singular *textual component*, an *inline directive* must be allowed to span over multiple matches.

6.0.1 Surrounding Inline Directives

An *inline directive* is a *surrounding inline directive* if its *identifier* contains syntactical features around a *content binding*.

When a *surrounding inline directive* is *processed*, processing begins anew over the *content binding* until the the part of the *identifier* after the *content binding* is *fully matched*, at which point the *resulting textual component* is ended and control is handed back to the *standard processing loop*.

6.0.2 Entity Inline Directives

An *inline directive* is an *entity inline directive* if its *identifier* does not contain any `binding`s and instead the *text* of the *resulting textual component* is entirely dependant on the *entity inline directive* specification.

When a *entity inline directive* is *processed*, the *resulting textual component* is ended once the *identifier* has been *fully matched*. Then control is handed back to the *standard processing loop*.

6.0.3 Compound Inline Directives

An *inline directive* is a *compound inline directive* if its *identifier* consists of multiple `binding`s the contents of which are in some form outputted to the *resulting textual component*.

6.1 Bold

Identifier **Bold**: `\*\*<content .*>\*\*`
Textual Component **Bold**: `font-weight: bold`

The bold *directive* is a *surrounding inline directive* that marks the *text* to belong to a *textual component* that sets the weight of the font to bold. Only the *text* held by the *content binding* is outputted to the *resulting textual component*.

Examples:

`not **bold** at all`	$\Rightarrow$	not **bold** at all
`and **some *things* are bad**`	$\Rightarrow$	and **some *things* are bad**

6.2 Italic

Identifier **Italic**: `//<content .*>//`
Textual Component **Italic**: `font-style: italic`

Italic is a *surrounding inline directive.* It marks the *text* to belong to a *textual component* that sets the style of the font to italic. Only the *text* held by the *content binding* is outputted to the *resulting textual component.*

Examples:
I `//really//` don't care. ⇒ I *really* don't care.

`//call/cc//` is important. ⇒ *call/cc* is important.

6.3 Underline

Identifier **Underline**: `__<content .*>__`
Textual Component **Underline**: `text-decoration: underline`

Underline is a *surrounding inline directive.* It marks the *text* to belong to a *textual component* that sets the style of the text to underline. Only the *text* held by the *content binding* is outputted to the *resulting textual component.*

Examples:
We `__must__` finish this. ⇒ We <u>must</u> finish this.

This `__CONSTANT_VALUE__` is variable. ⇒ This <u>CONSTANT_VALUE</u> is variable.

6.4 Strikethrough

Identifier **Strikethrough**: `\<-<content .*>-\>`
Textual Component **Strikethrough**: `text-decoration: strikethrough`

Strikethrough is a *surrounding inline directive.* It marks the *text* to belong to a *textual component* that sets the style of the text to strikethrough. Only the *text* held by the *content binding* is outputted to the *resulting textual component.*

Examples:
To Do: `<-nothing->` ⇒ To Do: ~~nothing~~

`<-Solve LOAD-TIME-VALUE problem->` ⇒ ~~Solve LOAD-TIME-VALUE problem~~

`<-Go -\> there->` ⇒ ~~Go -> there~~

6.5 Code

Identifier **Code**: `` `<content .*>` ``
Textual Component **Code**: `font-family: monospace`

Code is a *surrounding inline directive.* It marks the *text* to belong to a *textual component* that sets the font-family to monospace. Only the *text* held by the *content binding* is outputted to the *resulting textual component.* The code *directive* cannot contain any other *directives.*

Examples:

```
Call ``compile``                        ⇒   Call compile
Earmuffs ``*around*`` your specials.    ⇒   Earmuffs *around* your specials.
This: ``\``` is a backtick.             ⇒   This: ` is a backtick.
```

6.6 Dashes

Identifier **En-dash**: `--`
Identifier **Em-dash**: `---`
Textual Component **En-dash**: `display: en-dash`
Textual Component **Em-dash**: `display: em-dash`

The dashes are *entity inline directives*. If the *document* does not have direct support for dashes, a fallback character may be used when appropriate instead. In unicode encoded documents, this should be – (`U+2013`) for the en-dash and — (`U+2014`) for the em-dash.

Examples:

```
A game -- or gamble --- if you will.  ⇒   A game – or gamble — if you will.
```

6.7 Subtext

Identifier **Subtext**: `v\(<content .*>\)`
Textual Component **Subtext**: `vertical-align: sub`

Subtext is a *surrounding inline directive*. It marks the *text* to belong to a *textual component* that sets the style of the text to appear smaller and below the default text line. Only the *text* held by the *content binding* is outputted to the *resulting textual component*.

Examples:

```
This is an example v(just so you know)  ⇒   This is an example just so you know
```
```
 Sometimes you have to be discreet v(or so they say \(I wouldn't know\)).
```
⇒ Sometimes you have to be discreet or so they say (I wouldn't know)·

6.8 Supertext

Identifier **Supertext**: `^\(<content .*>\)`
Textual Component **Supertext**: `vertical-align: super`

Supertext is a *surrounding inline directive*. It marks the *text* to belong to a *textual component* that sets the style of the text to appear smaller and above the default text line. Only the *text* held by the *content binding* is outputted to the *resulting textual component*.

Examples:

```
This is a good example ^([citation needed])  ⇒   This is a good example [citation needed]
```
```
Nesting ^(supertext ^(is silly))  ⇒   Nesting supertext is silly
```

6.9 URL

Identifier **Url**: `<target ~a(~w|[+-.])*://(~w|[$-_.+!*'()&+,/:;=?@%#])+>`
Textual Component **Url**: `interaction: link; target: target`

URL is an *inline directive* that marks the *text* to belong to a *textual component* that sets its interaction to allow following to the URL target. The user must be presented with an action that allows them to follow to the URL target. The exact manner in which the target is followed as well as the way in which the action is presented are *implementation dependant*. The *text* of the *resulting textual component* must be exactly the same as that of the `target` *binding*.

Examples:
```
Come chat with us at irc://irc.libera.chat/%23shirakumo !
```
⇒ Come chat with us at irc://irc.libera.chat/%23shirakumo !

6.10 Compound

Identifier **Compound**: `''<content .>''\(<option .*>(, *<option .*>)*\)`
Textual Component **Compound**:

The compound *directive* is a *compound inline directive*. It determines its *style* dynamically by the additive combination of present options in the `option` binding. In the case where the style combination of two options conflicts, the style of the last option has priority.

Only the *text* held by the *content binding* is outputted to the *resulting textual component*. The `option` binding cannot contain any other *directives*.

An *implementation* must at least support the options specified in this section, but may add additional options the syntax and implications of which are completely *implementation dependant*. If an option is found that the *implementation* does not support or recognise, it is ignored and a *warning* may be *signalled*.

When a compound *directive* is *processed*, processing begins anew over the *content binding*. Once the `options` binding has been *fully matched*, the *resulting textual component* is ended and control is handed back to the *standard processing loop*.

6.10.1 Bold

Identifier **Compound-bold**: `bold`
Style **Compound-bold**: `font-weight: bold`

If given, this option marks the *style* to bold the *text*.

Examples:
```
Not ''again''(bold)!
```
⇒ Not **again**!

6.10.2 Italic

Identifier **Compound-italic**: `italic`
Style **Compound-italic**: `font-style: italic`

If given, this option marks the *style* to italicise the *text*.

Examples:

`This is ''really''(italic) important!` ⇒ This is *really* important!

6.10.3 Underline

Identifier **Compound-underline**: `underline`
Style **Compound-underline**: `text-decoration: underline`

If given, this option marks the *style* to be set to underline the *text*.

Examples:

`Solve it ''today''(underline)!` ⇒ Solve it <u>today</u>!

6.10.4 Strikethrough

Identifier **Compound-strikethrough**: `strikethrough`
Style **Compound-strikethrough**: `text-decoration: strikethrough`

If given, this option marks the *style* to be set to strikethrough the *text*.

Examples:

`''This is a good idea''(strikethrough).` ⇒ ~~This is a good idea~~.

6.10.5 Spoiler

Identifier **Compound-spoiler**: `spoiler`
Style **Compound-spoiler**: `display: hidden`

If given, this option marks the *style* to obscure the *text* in such a manner that the *user* must perform an *action* in order to reveal the *text*.

Examples:

`This is a ''secret''(spoiler)!` ⇒ This is a ███████!

6.10.6 Font

Identifier **Compound-font**: `font <font .+>`
Style **Compound-font**: `font-family: font`

If given, this option marks the *style* to change the font family. If the specified font is not available to the *user* for one reason or another, either no font change occurs, or an *error* is *signalled*. The *implementation* may make an effort to include the font in the *document* in such a way that it is not necessary for the user to have a copy of the font, but it is not required to.

Examples:

`''Comic sans''(font Comic Sans Ms) is a good font to annoy people.`

⇒ Comic sans is a good font to annoy people.

6.10.7 Color

Identifier **Compound-color**: `(color (<hex #.+>|<r ~n+> <g ~n+> <b ~n+>))|<name .+>`
Style **Compound-color**: `color: color`

   If given, this option marks the *style* to change the colour. The colour can be given in three ways:

1. Through a hexadecimal notation, contained in the `hex` *binding*. The *hexadecimal number* following the `#` must be exactly six *characters* long.

2. Through a red, green, blue component notation, contained in the `r`,`g`, and `b` *bindings*. Each of these bindings must contain a *decimal number* that may only range between 0 and 255. If the number lies outside this range, it is clamped to the nearest boundary.

3. Through an explicit colour name, contained in the `name` *binding*. The name must be *case insensitive*. The set of supported colour names is *implementation dependant*.

If the specified colour value is invalid or unknown to the *implementation* according to the above restrictions, an *error* is *signalled*. If the *document* does not support the specified colour, the *implementation* must choose an alternative colour that approximates the specified one as closely as possible.

Examples:

`This is ''blue''(blue).`	⇒	This is blue.
`''Magic!''(color #9D0ECC)`	⇒	Magic!
`Now in ''technicolor''(color 145 16 16).`	⇒	Now in technicolor.

6.10.8 Size

Identifier **Compound-size**: `(size (<point ~n+pt>|<em ~n+?(\.~n+?)?em>))|<name .+>`
Style **Compound-size**: `font-size: size`

   This option marks the *style* to change the font size. The size can be given in three ways:

1. Through a point value, contained in the `point` *binding*. The *real number* must be greater than zero.

2. Through an em value, contained in the `em` *binding*. The *real number* must be greater than zero. The font size is scaled according to the *real number* multiplied by the font size of the *textual component* one *level* below.

3. Through a name, contained in the `name` *binding*. The name must be *case insensitive*. At least the following names, corresponding to scaling factors, must be supported by the *implementation*:

   - Microscopic 0.25em
   - Tiny 0.5em
   - Small 0.8em
   - Normal 1.0em
   - Big 1.5em
   - Large 2.0em
   - Huge 2.5em
   - Gigantic 4.0em

   An implementation may support additional names, the exact sizing effects of which are *implementation dependant*.

If the specified size value is invalid or unknown to the *implementation* according to the above restrictions, no size change occurs.

Examples:

```
Oh ''shit!''(huge)
```
$\Rightarrow$ Oh shit!

```
In ''20pt.''(size 20pt)
```
$\Rightarrow$ In 20pt.

```
 Well ''uh, ''I don't know...''(size 0.5em)''(in size 0.8em)
```
$\Rightarrow$ Well uh, I don't know...

6.10.9 Hyperlink

Identifier **Compound-hyperlink**: `{url}|(#<internal .+>)|(link <external .+>)`
Style **Compound-hyperlink**: `interaction: link;target: target`

This option marks the *style* to set the interaction to allow following to the target. The user must be presented with an action that allows them to follow to the target. The exact manner in which the target is followed as well as the way in which the action is presented are *implementation dependant.* The target can be given in three ways:

1. As an URL, contained in the `target` *binding.* In this case the semantics are the same as for the *URL textual component.*

2. As an external reference, contained in the `external` *binding.* The exact semantics and allowed values for external references are *implementation dependant.*

3. As an internal reference, contained in the `internal` *binding.* The target is set to the position of the *textual component* associated with the *label* of the same name as the contents of the *binding.*

If the specified target is invalid or unknown to the *implementation* according to the above restrictions, no interaction change occurs.

Examples:

```
The ''hyperspec''(http://l1sp.org/cl/) is very useful.
```
$\Rightarrow$ The hyperspec is very useful.

```
And in ''part ''(#identifier-syntax)...
```
$\Rightarrow$ And in part 2...

```
I drew ''something''(~/drawings/test.jpg) today.
```
$\Rightarrow$ I drew something today.

6.11 Footnote-Reference

Identifier **Footnote-reference**: `\[<target ~n+>\]`
Textual Component **Footnote-reference**: `interaction:link;target:target;vertical-align:super`

The footnote-reference is a *surrounding inline directive* that marks the *text* to belong to a *textual component* that sets its interaction to allow following to the *label* with the name held by the *text* of the `target` *binding.* The user must be presented with an action that allows them to follow to the corresponding label. The exact manner in which the target is followed as well as the way in which the action is presented are *implementation dependant.* The *text* of the *resulting textual component* must be exactly the same as that of the entire *identifier.*

Examples:

```
Examples[1] are not authoritative.

[1] Examples are things like this.
```
$\Rightarrow$

Examples[1] are not authoritative.

—————————

1: Examples are things like this.

6.12   Newline

Identifier **Newline**: `-/-`
Textual Component **Newline**: `display: newline`

Newline is a *entity inline directive*. The following text in the *resulting textual component* should start on a new *line*.

Examples:

`This-/-and that` $\Rightarrow$ This
and that

Issues

issues/accidental directive invocation.mess

```
# Problem Description
Currently some inline directives are prone to accidental invocation, leading to
frustrating behaviour for users. Notable for this are:

~ Bold
| It's quite elegant: e^(i*pi)+1 = 0

Arguably for the above case the inline code block should be employed, but for the
italics there's no such excuse:

~ Italic
| There's problems/solutions to be found.

While the examples here only illustrate single uses which would not lead to a
successful match, it isn't hard to imagine that multiple separate uses like this
could occur in close vicinity.

# Solution Proposals
## Double Identifiers (Accepted)
Simply double the number of characters in the identifiers to disambiguate:

~ Bold
| This is now **bold**.

~ Italic
| This is //italic//.

## Strikethrough-like
~ Bold
| <*thing*>

~ Italic
| </thing/>

## Immediate Recurrence Escapes
If the content binding is empty, simply output the corresponding character instead
, or make the content have at least one character and introduce corresponding
entity inline directives. This would not fix the problem directly but would make
it a bit less awkward to type the given characters.

~ Bold
| *A times B* is: a**b.

This behaviour could also be seen as more surprising than less so, however.

# Issue Status
Resolved.
```

issues/line breaks.mess

Problem Description
Back when Markless was first designed, the question came up on how to accommodate
two rather different styles of line breaking. The styles are basically the
following:

~ Editor Style
| This is made for people who hate to resize their
| windows and thus manually insert line breaks
| everywhere to force the file into a specific width.
| Thus, this paragraph is made of a single line.

~ WYSIWYG Style
| This is made for people who are just typing stuff in a browser.
| They are not accustomed to weird coding practises, and thus expect this to have
two lines.

In order to account for both, Markless introduced "line modes" that can be
switched using an instruction. At the same time, it allowed for temporarily
switching between them for a single line by escaping the newline. Thus the
following:

~ Editor Style
| Should you ever want to explicitly insert a new
| line, you would do it with an escape \
| like that.

~ WYSIWYG Style
| Should you ever want to break something up over multiple lines, you'd do it \
| line this, without having to incur a newline.

This is all fine and good, but as soon as you start interpreting the escaped
newline as a way to continue a directive onto the next line, it stops making sense
. To illustrate:

~ Editor Style
| # Is this a header with a single line \
| or does it have two lines? What if you want to
| continue the header without incurring a new line?

~ WYSIWYG Style
| # The same problem here. Is this a header with a single line \
| or does it have two lines after all? What if you want the opposite?

Due to the inherent contradictory nature in both cases I conclude that continuing
a directive and controlling line breaks are two orthogonal features. Now the
question becomes: how do we deal with this?

Solution Proposals
Doubling the Line Modes
By having each line mode also specify the behaviour for the continuing of
directives onto the next line, each case could be addressed. This does not
particularly help with the intuitiveness of the entire problem however, as users
might come to expect different defaults. The explanation of what each mode means

exactly would also be complicated further.

Eliminating Editor Style
In WYSIWYG style the behaviour of "escaping ignores the newline entirely" is an
acceptable approach and makes intuitive sense. By eliminating editor style
altogether, this problem falls away.

Eliminating Singular Line Directives
By instead forcing singular line directives to become either guarded- or spanning
line directives this problem also falls away, as the behaviour is logically
defined in both cases. An example:

| ## This is a single header
| ## in both line modes.

Don't Allow Newlines
Another solution would be to disallow newlines in the content binding of a
singular line directive, thus again making every case unambiguous.

Absolute Escape Consistency (Accepted)
Escaping a newline never produces a newline in the resulting textual component and
 always continues the line onto the next one as if both lines were as one. This
means that \LF is the same as neither character existing at all, regardless of the
 line mode.

This also means it is impossible to emit newlines in editor mode. To reconcile
this, a new entity inline directive should be added that does this unquestioningly
.

Issue Status
Resolved.

issues/line directive simplicity.mess

Problem Description
One of the primary goals of Markless is to be relatively easy to parse. As a part
of this, a strong focus was put on making line directives fast to recognise. Many
of them thus follow a scheme of being identified by the first two characters on a
line. This is a very desirable property.

However, some of the directives do not currently use this scheme, and instead
require much more intricate parsing. A good example is the ordered line directive,
 which starts out with an arbitrary number. Since the number can easily grow
beyond 10, the two characters are quickly exceeded, and much more is needed to
parse.

Much worse still is the embed directive, which currently requires a scan of the
full line in order to determine whether a match occurred. This is strongly in
opposition to the intended goal.

Furthermore, while the spec currently says that "An implementation may optimise
the matching process of directives in the following manner: if the part of the
identifier before the content binding matches, then the whole identifier may be
considered matched" a more specific explanation of which match is necessary would

be good. Putting a hard constraint to the "first two characters" for line
directives would be optimal.

Alternate schemes should be devised for the ordered list, embed, and footnote
directives to work around this issue.

Solution Proposals
Footnote
1 (Accepted)
| [~d

Embed
1
| [~w

2 (Accepted)
| [

Ordered Line
1
| .~d

2 (Accepted)
| ~d(~d|.)

Issue Status
Resolved.

issues/compound stack depth.mess

Problem Description
The current syntax for the compound directive is roughly the following:

::
"foo"(bar)
::

This creates a problem when double quotes are used naturally as part of a
paragraph like so:

::
Catherine said "I don't know what to do," to which Jonathan snapped "You never do
!"
::

Due to it being allowed for a compound directive to contain another directive, the
 parser has to naively assume that each double quote starts a new compound
directive, until it either reaches the next part of the directive, ``"(`` or the
end of a block, at which point it can unwind again.

However, as lengthy paragraphs with many quotes are not uncommon, this can quickly
 lead the parser to reach a very deep directive stack, risking errors resulting
from a stack overflow, or bad parse behaviour from discarding stack frames.

Solution Proposals
Disallow Nesting
Disallowing the nesting of compound directives would swiftly solve it, as any eager parse of an opening double quote could be replaced by the next double quote encountered.

This comes at a high cost, however. Some text behaviours are only achievable through the compound directive, as bespoke inline directives do not exist for all compound options, making it impossible to achieve some behaviours.

Retroactive Stack Shifting
We impose a maximum compound nesting limit that is strictly lower than the stack limit. When another opening double quote is encountered and the nesting limit is reached, the stack is traversed downwards until a non-compound stack frame is reached. This frame is then removed from the stack by converting the associated component to a plain nesting component and re-emitting the double quote at its beginning.

This would allow all previous documents to keep working, and improve the worst-case behaviour of the implementation. However, it requires implementing rather specialised behaviour for this one directive, and it does not solve the problem at its root. If we nest other directives in between the eagerly parsed compound directives, the stack cap can sill be reached. Granted this is far less plausible to occur, but nevertheless possible.

Changing Opening Syntax (Accepted)
If we change the directive's identifier to something like ``''foo''(bar)`` instead, then accidental nesting is far less likely to occur. This would also mirror the other inline directives that all have two-character leading identifiers, rather than one.

The obvious downside of this approach is that it breaks all existing documents out there, which at this point there are quite a few of already.

Issue Status
Resolved.

Glossary

Action

Some form of interaction that a *user* viewing a *document* can perform.

Alphabetic

Any *character* that is one of the following:
```
a (U+61) b (U+62) c (U+63) d (U+64) e (U+65)
f (U+66) g (U+67) h (U+68) i (U+69) j (U+6A)
k (U+6B) l (U+6C) m (U+6D) n (U+6E) o (U+6F)
p (U+70) q (U+71) r (U+72) s (U+73) t (U+74)
u (U+75) v (U+76) w (U+77) x (U+78) y (U+79)
z (U+7A) A (U+41) B (U+42) C (U+43) D (U+44)
E (U+45) F (U+46) G (U+47) H (U+48) I (U+49)
J (U+4A) K (U+4B) L (U+4C) M (U+4D) N (U+4E)
O (U+4F) P (U+50) Q (U+51) R (U+52) S (U+53)
T (U+54) U (U+55) V (U+56) W (U+57) X (U+58)
Y (U+59) Z (U+5A)
```

Alphanumeric

Any *character* that is either *alphabetic* or *numeric*.

Applicable Directive

A *directive* is applicable if it is not a *disabled directive*.

Binding

A `binding` syntax rule, the content of which is the *string* it *matches*.

Case Insensitive

When both the lower- and upper-case representation of an *alphabetic character* are treated as *equivalent*.

Character

A singular entity as specified by an *encoding*.

Character Class

A specified set of *characters*.

Compound Inline Directive

An *inline directive* as specified in *compound inline directives*.

Conforming Document

A *document* that does not violate any of the requirements set forth by the *directives* outlined in this specification and can thus be properly *interpreted* by any *conforming implementation*.

Conforming Implementation

An *implementation* that fully and correctly adheres to all requirements laid down by this specification. An *implementation* may support additional features not described in this specification and still be conforming, as long as none of the features interfere with the *interpretation* of a *conforming document*.

Content Binding

> A *binding* with the `name` `content`.

Current Component

> The *textual component* on top of the parse stack.

Cursor

> A marker for where the *implementation* is in the inputting *document* during *interpretation*.

Decimal Number

> A sequence of *characters* that are *numeric* and thus form a mathematical number in base-10/decimal representation.

Directive

> A directive specifies what happens when the *implementation matches* a particular *identifier*. In particular, it may specify how the input *string* is *interpreted* into *text* in the *document*.

Disabled Directive

> 1) A *directive* on the *implementation*'s internal list of disabled directives. 2) A *directive* whose *identifiers* must not be recognised. 3) A *directive* that cannot be contained in the *current component*.

Document

> 1) The top-most *textual component* that is not contained in any other *textual component*. 2) A *string* to be interpreted into a *textual component* using rules outlined by *directives*.

Document Format

> A set of grammar and semantics to *interpret* the contents of a *document*.

Empty Line

> A *line* that only contains *whitespace* and a *newline*, or a sole *newline*.

Encoding

> A particular interpretation of a sequence of bytes into distinguishable *characters*.

Entity Inline Directive

> An *inline directive* as specified in *entity inline directives*.

Equivalent

> Two objects are considered equivalent, if they denote the same meaning or idea. In specific, two *characters* are equivalent, if they denote the same visual identity.

Error

> A message that indicates a problem with the *document* that makes it unable for the *interpretation* to proceed. When *signalled*, the *implementation* must abort *interpretation* without producing a resulting *document*.

Escaping

> If a *character* is preceded by \ (`U+5C`), it is "escaped" and its semantic meaning is suppressed.

Format

A particular representation of data.

Full Match

A *full match* occurs if a *match* occurs and the *cursor* is located immediately after the matched *string*.

Guarded Line Directive

A *line directive* as specified in *guarded line directives*.

Hexadecimal Number

A sequence of *characters* that are one of the following and thus form a mathematical number in base-16/hexadecimal representation.

```
0 (U+30) 1 (U+31) 2 (U+32) 3 (U+33) 4 (U+34)
5 (U+35) 6 (U+36) 7 (U+37) 8 (U+38) 9 (U+39)
a (U+61) b (U+62) c (U+63) d (U+64) e (U+65) f (U+66)
A (U+41) B (U+42) C (U+43) D (U+44) E (U+45) F (U+46)
```

Identifier

Some form of pattern or method by which a *string* is recognisable. More specifically, an *identifier* provides a means by which a *substring* can be distinguished from the rest of the *string*.

Identifier Specifier

A pattern in *identifier syntax* to specify the way in which the *identifier* can be recognised.

Implementation

Some form of program or system that implements the semantics of Markless.

Implementation Dependant

The exact implications are up to the *implementation* to decide, but must be clearly defined.

Inline Directive

A *directive* that can appear at any point within a *string* as specified in *inline directives*.

Instruction

An instance of the *instruction directive*.

Interpretation

The act of detecting *directives* and executing their effects on a *document*.

Label

1) A unique name within a *document* that is associated with a single *textual component* of the *document*. 2) A *textual component* that is associated through a *label*.

Level

A number representing the depth of a *directive* within the *document*. The level within any *directive* is one higher than the level the *directive* itself is at. The level of the *document* is always 0.

Line

1) Any sub-sequence within a *string* that is delimited by an *unescaped newline*. That is to say, a line always begins at either the beginning of the *string* or after the *newline*, and always ends at either the end of the *string* or with a *newline* that is not preceded by the \ (U+5C) *character*. 2) A *string* that is displayed in a single horizontal row.

Line Break Mode

A *variable* of the name `line-break` that specifies how *newline characters* are *interpreted* into the output *text* of the *document*. Only two values are permitted: `show`, `hide`.

Line Directive

A *directive* that spans one or more *lines* as specified in *line directives*.

Match

A *match* occurs if a *string* is exactly recognised by some specific pattern or method.

Metadata

Metadata is information that is not necessarily directly visible to the *user* in the resulting *document*.

Newline

Any *character* that represents that a new line should be started.

Numeric

Any *character* that is one of the following:
0 (U+30) 1 (U+31) 2 (U+32) 3 (U+33) 4 (U+34)
5 (U+35) 6 (U+36) 7 (U+37) 8 (U+38) 9 (U+39)

Page

Every *document* consists of one or more *pages*. A page is a container of some physical size within which *text* is displayed.

Parser State

A set of information that the *implementation* keeps during *interpretation*.

Position

A marker for where the *implementation* is in the resulting *document* during *interpretation*.

Processing

The act of evaluating a *directive* on a *matched string* and translating it into either effects on the *parser state* or *resulting textual components*.

Real Number

A sequence of *characters* as follows: One or more *numeric characters*, optionally followed by a . dot, followed by an arbitrary number of *numeric characters*. This forms a mathematical real number in base-10/decimal representation where the dot denotes the decimal point.

Resulting Textual Component

The *textual component* that the *directive* puts in place of the *identifier* in the *document*.

Signalling

The act by which an *implementation* can display a message to the *user*.

Singular Line Directive

A *line directive* as specified in *singular line directives*.

Spanning Line Directive

A *line directive* as specified in *spanning line directives*.

Specified Textual Component

A *textual component* that is declared in this specification.

Standard Processing Loop

The algorithm to *process* a *document* as described in *parser steps*.

String

A sequence of *characters*.

Style

A *style* is an attribute of a *textual component* that specifies how the *textual component* and its contents are supposed to be visually represented in the *document*.

Substring

A sequence of *characters* within a *string*.

Surrounding Inline Directive

An *inline directive* as specified in *surrounding inline directives*.

Text

Text is made up of a series of *strings* and *textual components*.

Textual Component

A section of *text* with specific visual *styling*, representation, and interaction properties.

User

Some entity —usually a human— that can view and interact with a *document*.

Variable

A name associated with some internal state of the *implementation*. See *set*.

Warning

A message that indicates a potential problem that occurred during *interpretation* that might cause the resulting *document* to appear wrong.

Whitespace

Any *character* that represents a horizontal gap. Examples include space, tab, zero-width space, etc.