Markless Specification

Contents

1 Preamble

Markless is a *document format* aimed to offer an unobtrusive and intuitive way to write plain-text *documents*. It is most easily comparable to Markdown but aims to avoid several of its kludges that conflict with the expectations inexperienced users have towards markup.

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.

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.
number                ::=  —  An integer.
```

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 the 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 UTF-8 *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* apply. 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

To be done:

- Line break behaviour
- Backslash escaping
- Parsing order
- Labels
- Parser states and switches

5 Line Directives

In order for a *directive* to be a *line directive*, its *identifier* must *match* the beginning of a *line* and either the end of a *line* or the beginning of a different *line*. Thus the *identifier* of each *line directive* only matches at 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*.

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*.

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*.

5.1 Paragraph

Identifier **Paragraph**:

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

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

The paragraph is the default *textual component* and acts as a fall-back. *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*. The paragraph *directive* can only contain *inline directives*.

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 another paragraph.
```

⇒

This is a paragraph
that spans multiple lines.

This is another paragraph.

```
Paragraph One
  Paragraph Two
```

⇒

Paragaph One

Paragraph Two

5.2 Blockquote

Identifier **Blockquote Header**:

```
\~ <content .+>
```

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 can only contain *inline directives*.

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.

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 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: c`

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!
```

⇒

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
```

⇒

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.
=====
Back to the show!
```

⇒

And now, for a brief break.

Back to the show!

5.6 Code block

Identifier **Code Block**:

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

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.

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 comment *directive* and as such it must not have any effect on the *document*.

The following instructions and their effect must be supported by an *implementation*.

`set <variable> <value>`	Sets an internal variable of the *implementation* to a certain value. An *implementation* may check the value for validity and *signal* an *error* if it is invalid.
`warn <message>`	Causes the *implementation* to *signal* a *warning* with the given message.
`error <message>`	Causes the *implementation* to *signal* an *error* with the given message.
`include <file>`	Literally splices the contents of the specified file into the *document* in place of this instruction. The *implementation* must carry on to *interpret* the newly spliced *text*.
`disable-directives <directive>*`	Adds the named directives to the list of *disabled directives*.
`enable-directives <directive>*`	Removes the named directives from the list of *disabled directives*.

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*.

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 an *surrounding inline directive* if its *identifier* is terminated by a `binding-reference` to a `binding` at the start of the *identifier*. The *resulting textual component* of a *surrounding inline directive* never contains the *strings matched* by the *identifier*'s starting `binding`. Instead, each *surrounding inline directive identifier* contains a *content binding* that matches all the *text* that the *resulting textual component* will contain.

6.0.2 Entity Inline Directives

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

6.0.3 Compound Inline Directives

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

6.1 Bold

Identifier **Bold**: `<start [*]+><content ![*].*><start>`
Textual Component **Bold**: `font-weight: bold`

The bold *directive* 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`	⇒	not **bold** at all
`and ***some *things* are bad***`	⇒	and **some *things* are bad**

6.2 Italic

Identifier **Italic**: `<start [/]+><content ![/].*><start>`
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**: `<start [_]+><content ![_].*><start>`
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 must finish this.
`This __CONSTANT_VALUE__ is variable.`	⇒	This CONSTANT_VALUE is variable.

6.4 Strikethrough

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

Strikethrough is an *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~~

6.5 Code

Identifier **Code**: `<start [`]+><content ![`].*><start>`
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.

6.6 Dashes

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

Em-dash is a *entity inline directive*. If the *document* does not have direct support for em-dashes, a fallback character may be used when appropriate instead. In unicode encoded documents, this should be —(U+2014).

Examples:

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

6.7 Subtext

Identifier **Subtext**: `v<start [(]+><content ![(].*><start>`
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 $_{\text{just so you know}}$

6.8 Supertext

Identifier **Supertext**: `^<start [(]+><content ![(].*><start>`
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 $^{\text{[citation needed]}}$

6.9 URL

Identifier **Url**: `<target ~w(~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*. The URL *directive* cannot contain any other *directives*.

Examples:

```
Come chat with us at irc://irc.freenode.net/%23lisp !
```
⇒ Come chat with us at irc://irc.freenode.net/%23lisp !

6.10 Compound

Identifier **Compound-content**: `<start ["]+><content !["].*><start>|<content ![ (]+>`
Identifier **Compound-options**: `((in|to)( <option .*>,)+)`
Identifier **Compound**: `{compound-content}{compound-options}`
Textual Component **Compound**:

The compound *directive* is a *compound inline directive*. It determines its *style* dynamically by the additive combination of present *compound-options*. 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 *compound-options identifier* 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, it is ignored.

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(in 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(in 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(in underline)!` ⇒ Solve it today!

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"(in 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(in spoiler)!` ⇒ This is a [redacted]!

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, no font change occurs. 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"(in 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, no colour change occurs. 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(in blue).`	⇒	This is blue.
`Magic!(in color #9D0ECC)`	⇒	Magic!
`Now in technicolor(in 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!"(in huge)  ⇒  Oh shit!
In "20pt."(in 20pt)  ⇒  In 20pt.
Well ""uh, "I don't know..."(in size 0.5em)""(in size 0.8em)
```

⇒ 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"(to http://llsp.org/cl/) is very useful.
```

⇒ The hyperspec is very useful.

```
And in "part 2"(to #identifier-syntax)...          ⇒ And in part 2...
I drew "something"(to ~/drawings/test.jpg) today.  ⇒ I drew something today.
```

Glossary

Action

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

Alphabetic

Any *character* that is one of the following:
`abcdefghijklmnopqrstuvwxyz`
`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

Alphanumeric

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

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`.

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.

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 when *signalled* causes the *implementation* to abort.

Format

A particular representation of data.

Guarded Line Directive

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

Hexadecimal Number

A sequence of *characters* that are one of `0123456789abcdefABCDEF` and thus form a mathematical number in base-16/hexadecimal representation.

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*.

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

Any sub-sequence within a *string* that is delimited by the *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*.

Line Break Mode

Specifies how *newline characters* are *interpreted* into the output *text* of the *document*.

Line Directive

A *directive* that spans one or more *lines*.

Match

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

Newline

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

Numeric

Any *character* that is one of the following:
`0123456789`

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 give feedback about the *interpretation* of the *document*.

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.

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*.

Warning

A message that when *signalled* 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.