vfmd

Markdown with a spec

The vfmd specification

vfmd is a variant of Markdown with an unambiguous specification of its syntax.

This document describes the specification for the vfmd syntax. It is intended to be read by someone implementing this specification to parse or otherwise programmatically interpret vfmd input. If you only intend to write a document using vfmd, please read the syntax guide instead.

Table of contents

- About this specification
  - Scope of this specification
  - Structure of this specification
  - Conventions
  - Definitions
- Identifying block-elements
  - The block-element line sequence
  - Type and extent of a block-element
- Interpreting block-elements
  - atx-style header
  - setext-style header
  - code block
  - blockquote
  - horizontal rule
  - unordered list
  - ordered list
  - paragraph
  - reference-resolution block
  - null block
- Identifying span-elements
  - Procedure for identifying span tags
  - Procedure for identifying link tags
  - Procedure for identifying emphasis tags
  - Procedure for identifying code-span tags
  - Procedure for identifying image tags
  - Procedure for detecting automatic links
  - Procedure for identifying HTML tags
- Additional processing
  - De-escaping
  - Processing text fragments

    - Processing for HTML output

- Extending the syntax

About this specification

Scope of this specification

This specification defines how a vfmd source document should be interpreted. Wherever special consideration is required in creating a HTML representation of the document, this specification also specifies the required HTML output. Special considerations that may be required for other output formats is not explicitly defined in this specification.

Structure of this specification

The specification is divided into the following major sections:

- About this specification: Defines terminologies and conventions used in this specification.
- Identifying block-elements: Defines how block-level syntax elements should be recognized.
- Interpreting block-elements: Defines how the block-level constructs, after being recognized, should be handled.
- Identifying span-elements: Defines how span-level syntax elements should be recognized and handled.
- Additional processing: Describes the processing that needs to be done on the non-markup parts of the document.
- Extending the syntax: Shows how this specification can be extended to support extensions to the core vfmd syntax.

Conventions

Regular expression conventions

This specification makes use of regular expressions to define the syntax. A regular expression appears as `/regularexpression/` in this specification - i.e. it appears in a code-span, enclosed between two forward slashes (like in Perl code).

The regular expressions follow the PCRE syntax in UTF mode. Specifically, the following should be considered when reading the regular expressions in this document:

1. A `\` character is used to escape any special character within a regular expression, irrespective of whether it has a special meaning at that position or not
2. A ` ` within a regular expression indicates a space character
3. A `\s` within a regular expression indicates a whitespace character

The regular expressions used in this specification do not use any extended regular expression syntax (e.g. min/max quantifiers, backreferences, etc.), and confirm to a regular grammar. This means that these regular expressions can be adapted to any other regular expression syntax as may be required for an implementation of this specification.

In this specification, when it is said that a string matches a regular expression, it is used in the meaning that a part or whole of the string matches the whole of the regular expression. Whenever the whole of the string needs to match the whole of the regular expression, that requirement is made explicit in the regular expression by starting it in `^` and ending it in `$`.

Definitions

Document

The input Markdown text is called the **document**.

The document contains Unicode text in UTF-8 encoding. If the document starts with the three bytes: `0xEF` `0xBB` `0xBF`, then those three bytes are interpreted as the UTF-8 Byte-Order-Mark and are filtered off and ignored. Any byte sequences in the input that are invalid in UTF-8 encoding are interpreted to be bytes in ISO-8859-1 encoding. Therefore, for the following discussion, the document is considered to not have the Byte-Order-Mark and not to have any invalid byte sequences.

Characters

A **character** is an atomic unit of text specified as a Unicode code point and encoded in UTF-8 encoding.

The document consists of a sequence of characters, where the characters may represent either markup or character data.

A U+0020 (SPACE) character is henceforth called a **space** character. Any character that is not a U+0020 (SPACE) character is henceforth called a **non-space** character.

A U+0009 (TAB) character is henceforth called a **tab** character.

The character sequence of a U+000D (CR) character followed by a U+000A (LF) character in the input shall be treated as a single U+000A (LF) character.

A U+000A (LF) character is called a **line break** character. Any character that is not a *line break* character is called a **non-line-break** character.

A **whitespace** character is one of the following characters: U+0009 (TAB), U+000A (LF), U+000C (FF), U+000D (CR) or U+0020 (SPACE)

A **string** is a sequence of zero or more characters.

Trimming a string means removing any leading or trailing whitespace characters from the string. For example, trimming `  yellow  ` yields `yellow`; trimming `green` yields `green`. Trimming a string that does not have any leading or trailing whitespace has no effect on the string. Trimming a string that is entirely composed of whitespace characters yields an empty (zero-length) string.

Simplifying a string means trimming the string, and in addition, replacing each sequence of internal whitespace characters in the string with a single space character. For example, simplifying `  Amazing  Maurice  ` yields `Amazing Maurice`; simplifying `educated  rodents` yields `educated rodents`.

Escaping a character in a string means placing a `\` (backslash) just before the character in the string, where the `\` used for escaping remains unescaped (i.e. the escaping `\` shall not itself be preceded by an unescaped `\`).

A **quoted string** is a string that consists at least two characters and either

1. begins with an unescaped `'` (single quote) character, and ends with an unescaped `'` character, and does not contain any other instance of an unescaped `'` character

   or

2. begins with an unescaped `"` (double quote) character, and ends with an unescaped `"` character, and does not contain any other instance of an unescaped `"` character

The substring of the quoted string that excludes the first and the last character of the quoted string is called the **enclosed string** of the quoted string.

Lines

A **line** is a sequence of non-line-break characters.

When we split the document on line breaks, we get *lines*. The document can then be seen as a sequence of *lines*, separated by line breaks.

If a line contains no characters, or if all characters in the line are space or tab characters, that line is called a **blank line**.

The lines in the document shall be preprocessed to expand each tab character in the line to a certain number of space characters, as specified below:

1. Let *p* be the position of the tab character in the line, after all previous tab characters in the line have been "expanded".

   For the first character of a line, the position of the character is considered to be 0; for the second character of the line, the position of the character is considered to be 1, and so on.

2. Compute *n* as: (4 - (*p* % 4)), where '%' is the modulo operator

3. "Expand" the tab character by replacing it with *n* number of space characters

Since all the lines in a document are tab-expanded as specified above, for the following discussion, the document is considered not to have any tab characters.

To identify the block-elements in the document, the document is seen as a sequence of lines.

Identifying block-elements

Given a sequence of lines, henceforth called the **input line sequence**, we need to identify the block-elements in the input, identify the type of the block-elements and identify which sub-sequence of lines in the input correspond to which block-element.

The block-element line sequence

The line at which a block-element begins is called a **block-element start line**. The line at which a block-element ends is called a **block-element end line**. The sub-sequence of lines starting from the *block-element start line* and ending at the *block-element end line*, both inclusive, constitute the **block-element line sequence**.

Every *block-element end line* in the input line sequence is immediately followed by a *block-element start line*, unless the *block-element end line* is the last line in the input line sequence. The first line in the input line sequence is a *block-element start line*, and the last line in the input line sequence is a *block-element end line*.

The idea is to break the input line sequence into a series of *block-element line sequences*. Each *block-element line sequence* shall consist of only the lines that correspond to a particular block-element.

When a *block-element line sequence* is obtained by breaking up an input line sequence, the input line sequence is said to be the *parent line sequence* of the *block-element line sequence*.

Type and extent of a block-element

The type of the block-element is determined based on the block-element start line and, in some cases, the line following the block-element start line. The line at which the block should end (i.e. the corresponding block-element end line) is determined based on the block-element start line and subsequent lines.

We define the following regular expression patterns:

- **unordered list starter pattern**: `/^( *[\*\-\+] +)[^ ]/`

Example: `  *`

- **ordered list starter pattern**: `/^( *([0-9]+)\. +)[^ ]/`

Example: `  1.`

- **horizontal rule pattern**: `/^ *((\* *\* *\* *[\* ]*)|(\- *\- *\- *[\- ]*)|(_ *_ *_ *[_ ]*))$/`

Any line that matches the *horizontal rule pattern* would be composed of just asterisks (minimum three) and space characters, or just underscores (minimum three) and space characters, or just dashes (minimum three) and space characters.

Examples:
`*****`
`-- -- --`

The following rules are to be followed in determining the type of the block-element and the block-element end line:

1. If the block-element start line is a blank line, then the block element is of type **null block**. The same line is the block-element end line.

2. If the block-element start line does not begin with four or more consecutive space characters, and satisfies both the following conditions:

    1. The block-element start line matches the regular expression pattern:

        `/^ *\[(([^\\\[\]\!]|\\.|\![^\[])*((\!\[([^\\\[\]]|\\.)*\](\[([^\\\[\]]|\\.)*\])?)?([^\\\[\]]|\\.)*)*)\] *:(.*)`

        Examples:
        `[ref id]:` + *ref-value-sequence*
        `[ref \[escaped brackets\] id]:` + *ref-value-sequence*
        `[![image]]:` + *ref-value-sequence*
        `[![image][image ref]]:` + *ref-value-sequence*

        The matching substring for the first parenthesized subexpression in the pattern shall be called the *unprocessed reference id string*. The matching substring for the last parenthesized subexpression in the pattern shall be called the *ref-value-sequence*.

    2. The *ref-value-sequence* matches the regular expression pattern:

        `/^ *([^ \<\>]+|\<[^\<\>]*\>)( .*)?$/`

        Examples:
        `url` + *ref-definition-trailing-sequence*
        `<url>` + *ref-definition-trailing-sequence*

        The matching substring for the first parenthesized subexpression in the pattern shall be called the *unprocessed url string*. The matching substring for the last parenthesized subexpression in the pattern shall be called the *ref-definition-trailing-sequence*.

then the block-element is of type **reference-resolution block**.

If all the following conditions are satisfied:

1. The *ref-definition-trailing-sequence* does not contain any non-space characters
2. The block-element start line is not the last line in the input line sequence
3. The block-element start line is immediately followed by a succeeding line that matches the regular expression pattern `/^ +("([^"\\]|\\.)*"|'([^'\\]|\\.)*'|\(([^\\\(\)]|\\.)*\)) *$/`

   Examples:
   `"Title"`
   `(Title)`
   `'A "title" in single quotes'`
   `"Title with \"quotes\""`

then the block-element end line is the line that immediately follows the block-element start line; else, the block-element end line is the same line as the block-element start line.

3. If none of the above conditions apply, and if the block-element start line is not the last line in the input line sequence, and is immediately followed by a succeeding line that matches the regular expression pattern `/^(-+|=+) *$/`, then the block-element is said to be of type **setext-style header**, and the succeeding line is said to be the block-element end line.

4. If none of the above conditions apply, and if the block-element start line begins with four or more consecutive space characters, it signifies the start of a block-element of type **code block**. The block-element end line is the next subsequent line in the input line sequence, starting from and inclusive of the block-element start line, that is immediately succeeded by a succeeding line that satisfies one of the following conditions:

   1. The succeeding line is not a blank line, and it does not begin with four or more consecutive space characters

      (or)

   2. The succeeding line is a blank line, and is immediately followed by a line that does not begin with four or more consecutive space characters

   If no such block-element end line is found, the last line in the input line sequence is the block-element end line.

5. If none of the above conditions apply, and if the first character of the block-element start line is a `#` character, it signifies the start of a block-element of type **atx-style header**. The same line is the block-element end line.

6. If none of the above conditions apply, and if the first non-space character in the block-element start line is a `>` character, then the block-element is of type **blockquote**. The block-element end line is the next subsequent line in the input line sequence, starting from and inclusive of the block-element start line, that satisfies one of the following conditions:

   1. The line is a blank line and is immediately succeeded by another blank line

      (or)

   2. The line is a blank line and is immediately succeeded by a succeeding line that begins with four or more consecutive space characters

      (or)

   3. The line is a blank line that is immediately succeeded by a succeeding line, and the first non-space character in the succeeding line is not a `>` character

      (or)

   4. The line is not a blank line and is immediately succeeded by a succeeding line that statisfies all of the following conditions:

      1. The succeeding line does not begin with four or more consecutive space characters
      2. The succeeding line matches the horizontal rule pattern

   If no such block-element end line is found, the last line in the input line sequence is the block-element end line.

7. If none of the above conditions apply, and if the block-element start line matches the horizontal rule pattern, then the line forms a block-element of type **horizontal rule**.

   The block-element end line is the same as the block-element start line.

8. If none of the conditions specified above apply, and if the block-element start line matches the unordered list starter pattern (i.e. the regular expression `/^( *[\*\-\+] +)[^ ]/`) then the block-element is of type **unordered list**. The matching substring for the first and only parenthesized subexpression in the pattern is called the *unordered list starter string*. The number of characters in the *unordered list starter string* is called the *unordered-list-starter-string-length*.

   For example, consider the following block-element start line (which has three space characters, followed by an asterisk, followed by two space characters, followed by the word "Peanuts"):

   ```
   *  Peanuts
   ```

   The *unordered list starter string* in the above example consists of the first 6 characters of the line, i.e. the entire part before the word "Peanuts". The *unordered-list-starter-string-length* is 6.

   The block-element end line is the next subsequent line in the input line sequence, starting from and inclusive of the block-element start line, that satisfies one of the following conditions:

   1. The line is a blank line and is immediately succeeded by another blank line

      (or)

   2. The line is a blank line and is immediately succeeded by a succeeding line that satisfies all of the following conditions:

      1. The succeeding line does not start with the *unordered list starter string* seen in the block-element start line
      2. The first *unordered-list-starter-string-length* characters of the succeeding line include non-space characters

      (or)

   3. The line is not a blank line and is immediately succeeded by a succeeding line that statisfies all of the following conditions:

      1. The succeeding line does not start with the *unordered list starter string* seen in the block-element start line
      2. The first *unordered-list-starter-string-length* characters of the succeeding line include non-space characters
      3. The succeeding line does not begin with four or more consecutive space characters
      4. The succeeding line matches the unordered list starter pattern, or matches the ordered list starter pattern, or matches the horizontal rule pattern

   If no such block-element end line is found, the last line in the input line sequence is the block-element end line.

9. If none of the conditions specified above apply, and if the block-element start line matches the ordered list starter pattern (i.e. the regular expression `/^( *([0-9]+)\. +)[^ ]/`) then the block-element is of type **ordered list**. The length of the matching substring for the first (i.e. outer) parenthesized subexpression in the pattern is called the *ordered-list-starter-string-length*.

   For example, consider the following block-element start line (which has three space characters, followed by the number '1', followed by a dot, followed by two space characters, followed by the word "Peanuts"):

   ```
   1.  Peanuts
   ```

   The *ordered-list-starter-string-length* in the above example is 7.

   The block-element end line is the next subsequent line in the input line sequence, starting from and inclusive of the block-element start line, that satisfies one of the following conditions:

   1. The line is a blank line and is immediately succeeded by another blank line

      (or)

   2. The line is a blank line and is immediately succeeded by a succeeding line that satisfies all of the following conditions:

      1. The succeeding line does not match the ordered list starter pattern
      2. The first *ordered-list-starter-string-length* characters of the succeeding line include non-space characters

      (or)

   3. The line is not a blank line and is immediately succeeded by a succeeding line that satisfies all of the following conditions:

      1. The succeeding line does not match the ordered list starter pattern

2. The first *ordered-list-starter-string-length* characters of the succeeding line include non-space characters
3. The succeeding line does not begin with four or more consecutive space characters
4. The succeeding line either matches the unordered list starter pattern, or matches the horizontal rule pattern

If no such block-element end line is found, the last line in the input line sequence is the block-element end line.

10. If none of the above conditions apply, then the block-element start line marks the start of a block-element of type **paragraph**.

    In order to find the block-element end line of a paragraph, we need to make use of a code-span detector and a HTML parser, configured to act in tandem as specified below:

    1. When the code-span detector is *deactivated*, it shall ignore all input passed to it. Similarly, when the HTML parser is *deactivated*, it shall ignore all input passed to it. After being *deactivated*, both the code-span detector and the HTML parser can be *activated* again, on which they shall resume processing new input. Any input ignored when *deactivated* shall remain ignored.
    2. When the code-span detector enters the "within a code-span" state, it shall *deactivate* the HTML parser. When the code-span detector exits the "within a code-span" state, it shall *activate* the HTML parser and immediately pass the input `<code><code/>` to the HTML parser.
    3. When the HTML parser enters the "within a quoted attribute value of a HTML tag" state, it shall *deactivate* the code-span detector, and when it exits the "within a quoted attribute value of a HTML tag" state, it shall *activate* the code-span detector.

    First, we reset the code-span detector and the HTML parser to their initial state. Then, starting from (and inclusive of) the block-element start line, we input each line to the code-span detector and to the HTML parser as described below:

    For each line:

    1. Any escaped `<` characters in the line are replaced with the string `&lt;` and the result is called the *processed line*
    2. For each character in the *processed line*, the character is first given as input to the code-span detector, and then given as input to the HTML parser.

       That is, we pass the first character to the code-span detector, then pass the first character to the HTML parser, then pass the second character to the code-span detector, then pass the second character to the HTML parser, and so on, till we reach the end of the *processed line*.
    3. A line break character is given as input to the code-span detector, and then given as input to the HTML parser.
    4. We observe the state of the HTML parser. Of the many possible states of a HTML parser, we are only interested in the HTML parser states relevant to finding the end of a paragraph.

    The block-element end line is the next subsequent line in the input line sequence, starting from and inclusive of the block-element start line, that satisfies all the following conditions:

    1. At the end of feeding all lines till (and inclusive of) this line, to the HTML parser, all the following conditions are satisfied:
       1. The HTML parser state is not "within a HTML tag"
       2. The HTML parser state is not "within a quoted attribute value of a HTML tag"
       3. The HTML parser state is not "within a HTML comment"
       4. The HTML parser state is not "within the contents of a well-formed verbatim HTML element"
    2. The line is a blank line, or both the following conditions are satisfied:
       1. The line is immediately succeeded by a succeeding line that does not begin with four or more consecutive space characters, and satisfies at least one of the following conditions:
          1. The succeeding line matches the horizontal rule pattern

             (or)

          2. The leftmost non-space character in the succeeding line is a `>` character, and the input line sequence is a blockquote-processed line sequence

             (or)

3. The succeeding line matches the ordered list starter pattern, and the input line sequence is a list-item-processed line sequence

   (or)

4. The succeeding line matches the unordered list starter pattern, and the input line sequence is a list-item-processed line sequence

2. The HTML parser, after having consumed all lines till (and inclusive of) this line, has not encountered any HTML tag with tag name as either a verbatim-html-starter-tag-name or a verbatim-html-container-tag-name

If no such block-element end line is found, the last line in the input line sequence is the block-element end line.

Using the above rules, the input line sequence is broken down into a series of block-element line sequences, and the block-element type of each block-element line sequence is identified.

An implementation can extend the core vfmd syntax to support additional syntax elements. For every additional block-level syntax element, the implementation shall insert a rule in the above rule list, at a position at which it makes sense to recognize the particular construct. Given a block-element start line, the rule shall identify whether the line is the beginning of the said block-level construct, and shall determine till what line in the input line sequence the block-level construct extends. Doing so will identify the block-element line sequence for the block-level construct. The implementation can decide how the block-element line sequence should be interpreted and output.

Code-span detector

The code-span detector is used to detect code-spans while looking for the end of a paragraph block. The code-span detector is always in one of the following two states:

1. Within a code-span
2. Not within a code-span

The code-span detector takes one character of input at a time, and maintains internal state information in addition to the two externally visible states mentioned above. The following internal state information is preserved across multiple characters of input:

1. **backticks-count**: The number of `` ` `` characters in the current backtick sequence
2. **open-backticks-count**: The number of `` ` `` characters in the opening tag of the current code span

When the code-span detector is **reset** or **initialized**, the backticks-count and open-backticks-count are set to 0, and the state of the code-span detector is set to "Not within a code-span".

For each character that is input to the code-span detector, the following is done

1. Let *input-character* be the character that is input
2. If the *input character* is a `` ` `` character, do the following:
   1. If the state of the code-span detector is "Not within a code-span", and if the *input character* is an unescaped `` ` `` character, then set the state of the code-span detector to "Within a code-span"
   2. Increment backticks-count
3. If the *input character* is not a `` ` `` character, do the following:
   1. If backticks-count is greater than 0, then set *is-end-of-backtick-sequence* to *true*

      If backticks-count is equal to 0, then set *is-end-of-backtick-sequence* to *false*

   2. If *is-end-of-backtick-sequence* is true, do the following:
      1. If open-backticks-count is greater than 0, then set *is-in-code-span* to *true*.

         If open-backticks-count is equal to 0, then set *is-in-code-span* to *false*

      2. If *is-in-code-span* is *false*, then do the following:
         1. Set open-backticks-count to backticks-count
      3. If *is-in-code-span* is *true*, and if open-backticks-count is equal to backticks-count, then do the following:
         1. Set open-backticks-count to 0
         2. Set the state of the code-span detector to "Not within a code-span"

4. Set backticks-count to 0

Note that this procedure intentionally does not take care of unclosed code spans. The code span detector is used solely for finding the end of a paragraph and it is not necessary to handle unclosed code spans in it.

HTML parser states relevant to finding the end of a paragraph

When looking for the block-element end line for a block-element of type paragraph, we make use of a HTML parser. Of the many possible states of a HTML parser, we are interested in only some of the states. This section enumerates and describes the parser states that are of interest in this context.

We define a **verbatim HTML element** to be one of these HTML elements: `pre`, `script`, `tag`. We define a **non-verbatim HTML element** to be any HTML element other than a *verbatim HTML element*.

The relevant states of the HTML parser when looking for the block-element end line for a block-element of type paragraph are:

1. Within a quoted attribute value of a HTML tag

   This is the state within attribute values enclosed in either single or double quotes. For example, this is the state at the end of the first line below:

   ```
   <img src="image.png" title="Title spanning
   multiple lines">
   ```

2. Within a HTML tag (open / close / self-closing tag)

   For example, this is the state at the end of the first line below:

   ```
   <div id="div1"
   >
   ```

3. Within a HTML comment

   For example, this is the state at the end of the first line below:

   ```
   <!-- Insert illuminating comment here
   -->
   ```

4. Within the contents of a well-formed verbatim HTML element

   For example, this is the state at the end of the first line below:

   ```
   This open <pre> tag has a
   corresponding close </pre> tag
   ```

5. Within the contents of a not-well-formed verbatim HTML element (i.e. after the open tag of an unclosed or not-properly-closed verbatim HTML element)

   For example, this is the state at the end of the first line below:

   ```
   This open <pre> tag does not have a
   corresponding close tag in the document
   ```

6. Within the contents of a non-verbatim HTML element (well-formed or not)

   For example, this is the state at the end of the first line below:

   ```
   Outside <div> Inside
   Inside </div> Outside
   ```

7. Outside of any HTML element or comment

   For example, this is the state at the end of the first line below:

   ```
   Outside <div> Inside </div> Outside
   Outside
   ```

Note that a HTML parser can know whether a `<` marks the start of a HTML construct or not only after it has seen the rest of the text. For example, if a matching `>` is not found, the first `<` does not indicate the start of a HTML construct at all. As another example, if the `<` is immediately followed by a `!--` and later followed by a `-->`, the `<` marks the start of a HTML comment. Consequently, the HTML parser should lookahead as many characters as may be necessary and return the appropriate state considering all these possibilities.

Also note that a HTML parser can know whether a HTML element is well-formed or not, only after encountering an end tag. So, after consuming only part of the input, the HTML parser might not know whether it's in the "within the contents of a well-formed verbatim HTML element" state, or it's in the "within the contents of a not-well-formed verbatim HTML element" state.

In order to be able to get the state of the HTML parser correctly as defined above, it is suggested that an implementation follow one of the following methods:

1. **Multiple-pass parsing:**

   Once the block-element start line is identified as belonging to a paragraph block, the entire sequence of lines starting from the block-element start line till the end of the input line sequence is passed to a HTML parser, which identifies all HTML tags, their positions and whether they are well-formed (i.e. whether they have corresponding opening/closing tags, as applicable).

   In the second pass, the information gathered in the first pass is used to drive the procedure to detect the end of the paragraph.

   This method is possibly simpler to implement.

2. **Backtracking:**

   When finding the HTML state at the end of a line requires data from subsequent lines, the implementation can assume well-formedness and proceed with processing the subsequent lines. If the assumption turns out to be wrong, it should backtrack to the original state and continue as if it is not well-formed.

   For example, if a line ending with `<pre>` is encountered, we assume that the opening tag will get closed subsequently and proceed with processing the subsequent lines. If no matching `</pre>` is found in the input line sequence, we have to backtrack to the first blank line after the opening `<pre>` and end the paragraph there.

   This method is more complex to implement, but will also be more optimal in terms of performance, especially for well-formed input.

Interpreting block-elements

This section assumes that the input line sequence has been broken down into a series of block-element line sequences, and that the block-element type of each block-element line sequence has been identified. For a given block-element type, the procedure to interpret a block-element line sequence is discussed in this section.

null block

The block-element line sequence for a null block element shall have a single blank line.

A null block does not result in any output.

reference-resolution block

The block-element line sequence for a reference-resolution block shall have either a single line or two lines.

The reference-resolution block does not result in any output by itself. It is used to resolve the URLs of reference-style links and images in the rest of the document.

When the block-element start line for the reference-resolution block was identified, the following values would have been identified as well:

- *unprocessed reference id string*
- *unprocessed url string*
- *ref-definition-trailing-sequence*

The *unprocessed reference id string* is simplified to obtain the *reference id string*.

Any `<`, `>` or whitespace characters in the *unprocessed url string* are removed, and the resultant string is called the *link url string*.

In case the block-element line sequence contains a single line, the *ref-definition-trailing-sequence* shall be called the *title container string*.

In case the block-element line sequence contains two lines, the second line shall be called the *title container string*.

If the *title container string* matches the regular expression pattern `/^\(((?:[^\\()]|\\.)*)\)/`, then the matching substring for the first (i.e. outer) parenthesized subexpression in the pattern is called the *link title string*.

If the *title container string* begins with a quoted string, the enclosed string of the quoted string is called the *link title string* and the rest of the *title container string* is ignored.

The *reference id string* is said to be associated with the *link url string* and the *link title string* (if a *link title string* was found). A new entry is added to the link reference association map with the *reference id string* as the key, and the *link url string* and the *link title string* as values, unless the link reference association map already has an entry with the *reference id string* as the key.

The **link reference association map** is an associative array that contains data from all the reference-resolution blocks in the document, including those that occur within blockquotes and lists. It helps in mapping a *reference id* to the *link url* and *link title* that the *reference id* represents. It is used in the procedure for identifying link tags and in the procedure for identifying image tags to resolve a reference id to a link url and, if available, a link title. All lookups in the *link reference association map* are made case-insensitively.

atx-style header

The block-element line sequence for an atx-style header shall have a single line that starts with a `#` character.

The single line in the block-element line sequence shall match one of the following regular expression patterns:

1. With header text: `/^(#+)(.*[^#])#*$/`

   Examples:

   ```
   ## Subheading 1
   ### Third-level *heading*
   ####Fourth-level####
   ##   Subheading #2   ####
   ###### Six hashes
   ####### Seven hashes
   ######## Eight '#'es
   ```

   The length of the matching substring for the first parenthesized subexpression is the heading level, subject to a maximum of 6.

   The matching substring for the second parenthesized subexpression shall be trimmed to give a *header text run*. The result of interpreting the *header text run* as a text span sequence shall form the content of the header element.

   For example, the HTML outputs for the above expressions are:

   ```
   <h2>Subheading 1</h2>
   <h3>Third-level <em>heading</em></h3>
   <h4>Fourth-level</h4>
   <h2>Subheading #2</h2>
   <h6>Six hashes</h6>
   <h6>Seven hashes</h6>
   <h6>Eight '#'es</h6>
   ```

2. Without header text: `/^(#+)$/`

   The length of the matching substring for the first parenthesized subexpression is the heading level, subject to a maximum of 6. The header text is empty.

   For example, if the complete line reads:

   ```
   #####
   ```

   then the corresponding HTML output shall be:

   ```
   <h5></h5>
   ```

setext-style header

The block-element line sequence for a setext-style header shall have exactly two lines, with the second line beginning with either a `-` character or a `=` character.

The first line shall be trimmed to give a *header text run*. The result of interpreting the *header text run* as a text span sequence shall form the content of the header element.

If the second line starts with the `=` character, the heading level shall be 1. If the second line starts with the `-` character, the heading level shall be 2. No other heading levels are possible in a setext-style header.

For example, consider the following pairs of lines:

```
Level One
=========

Another *Level One*
===================

Level   Two
-----------

Another level two
-----------------
```

The corresponding HTML outputs for the above lines are:

```
<h1>Level One</h1>

<h1>Another <em>Level One</em></h1>

<h2>Level   Two</h2>

<h2>Another level two</h2>
```

code block

Each line in the block-element line sequence for a code block element shall either be a blank line or a line beginning with four or more space characters.

For each line in the block-element line sequence, the leading four space characters are removed, if present, and the resulting sequence of lines, separated by line breaks, forms the content of the code block. For HTML output, the content of the code block should be html-code-escaped.

For example, if the following sequence of lines form the code block element:

```
    int main() {
        return 42;
    }
```

The corresponding HTML output shall be:

```
<pre><code>int main() {
    return 42;
}
</code></pre>
```

blockquote

The block-element line sequence for a blockquote element shall have one or more lines, some of which might have the `>` character as the first non-space character in the line.

If the last line in the block-element line sequence is a blank line, the last line is ignored.

Each line in the block-element line sequence is processed to produce a modified sequence of lines, called the *blockquote-processed line sequence*. The following processing is to be done for each line:

1. If the line matches the regular expression `/^ *> /`, then the part of the line that matches the said regular expression shall be removed from the line
2. If the pattern in (1) above is not satisfied, and if the line matches the regular expression `/^ *>/`, then the part of the line that matches the said regular expression shall be removed from the line

The blockquote-processed line sequence obtained this way can be considered as the input line sequence for a sequence of block-elements nested within the blockquote. The result of interpreting that input line sequence further into block-elements shall form the content of the blockquote element.

For example, consider the following block-element line sequence:

```
  > In Perl, a Hello World is
  > written as follows:
  >
  >     print "Hello World!\n";
```

After processing each line in the above block-element line sequence, the blockquote-processed line sequence obtained is as follows:

```
In Perl, a Hello World is
written as follows:

    print "Hello World!\n";
```

When we treat the blockquote-processed line sequence as an input line sequence, we can recognize nested block elements in it of type paragraph and code block. The HTML equivalent for the lines in the blockquote-processed line sequence is as follows:

```
<p>In Perl, a Hello World is
written as follows:</p>

<pre><code>print "Hello World!\n";
</code></pre>
```

Therefore, the HTML equivalent for the given block-element line sequence is:

```
<blockquote>
<p>In Perl, a Hello World is
written as follows:</p>

<pre><code>print "Hello World!\n";
</code></pre>
</blockquote>
```

horizontal rule

The block-element line sequence for a horizontal rule element shall have a single line that is composed entirely of either `*`, `-` or `_` characters, along with optional space characters.

The output shall consist of a horizontal rule.

For example:

```
Last line of a paragraph.

* * *

First line of a paragraph.
```

The corresponding HTML output shall be:

```
<p>Last line of a paragraph.</p>

<hr/>

<p>First line of a paragraph.</p>
```

unordered list

The block-element line sequence for an unordered list block shall have one or more lines.

The first line in the block-element line sequence would match the unordered list starter pattern (i.e. the regular expression `/^( *[\*\-\+] +)[^ ]/`). The matching substring for the first and only parenthesized subexpression in that pattern is called the *unordered list starter string*. The number of characters in the *unordered list starter string* is called the *unordered-list-starter-string-length*.

We first divide the block-element line sequence into a series of *unordered list item line sequences*. The lines in a particular *unordered list item line sequence* correspond to one list item in the list.

Every line in the block-element line sequence that starts with the *unordered list starter string* is called an *unordered list item start line*. Each *unordered list item start line* signifies the beginning of a new *unordered list item line sequence*. An *unordered list item line sequence* consists of the sequence of lines starting from (and inclusive of) an *unordered list item start line*, and ending at (and excluding) the next subsequent *unordered list item start line*. If there is no subsequent *unordered list item start line*, the *unordered list item line sequence* ends at the last line of the block-element line sequence.

We have now divided the block-element line sequence into a series of *unordered list item line sequences*. The first line of each *unordered list item line sequence* starts with the *unordered list starter string*.

Each line in the unordered list item line sequence is processed to produce a modified sequence of lines, called the *unordered-list-item-processed line sequence*. The following processing is to be done for each line:

1. If the line is the first line of the *unordered list item line sequence*:

The line would start with the *unordered list starter string*. The *unordered list starter string* shall be removed from the beginning of the line.

2. If the line is not the first line of the *unordered list item line sequence*:

   The line would start with zero or more space characters. The leading space characters, if any, should be removed as given below:

   1. If the number of leading space characters exceeds the *unordered-list-starter-string-length*, the number of leading space characters removed should be equal to the *unordered-list-starter-string-length*.
   2. If the number of leading space characters is less than or equal to the *unordered-list-starter-string-length*, all the leading space characters should be removed.

Each unordered-list-item-processed line sequence obtained this way can be considered as the input line sequence for a sequence of block-elements nested within the list item. The result of interpreting that input line sequence further into block-elements shall form the content of the list element.

An unordered-list-item-processed line sequence has certain properties that are useful in determining how the paragraph block-elements (if any) contained within the unordered-list-item-processed line sequence should be handled.

The list elements so obtained are combined into a sequence to form the complete unordered list in the output.

For example, consider the following block-element line sequence:

```
* First item 1

* Second item 1
Second item 2

      Code block

* Third item 1

    * Nested item 1
```

The *unordered list starter string* for the above example is an asterisk followed by a single space character. The *unordered-list-starter-string-length* is 2.

The 1st, 3rd and 8th lines in the block-element line sequence start with the *unordered list starter string*, and are therefore *unordered list item start lines*. (The 10th line does contain the *unordered list starter string*, but does not start with the *unordered list starter string*, so it's not an *unordered list item start line*.) Therefore, there are three *unordered list item line sequences* in the above example, as follows:

1. The lines 1 and 2 form the first *unordered list item line sequence*
2. The lines 3-7 form the second *unordered list item line sequence*
3. The lines 8-10 form the third and last *unordered list item line sequence*

Each *unordered list item line sequence* is then processed to obtain the *unordered-list-item-processed line sequence*. When we treat each *unordered-list-item-processed line sequence* as an input line sequence, we can recognize nested block elements in it.

The first *unordered list item line sequence* looks like:

```
* First item 1
```

To obtain the corresponding *unordered-list-item-processed line sequence*, we need to remove the *unordered list starter string* from the beginning of the first line. Since the second line is a blank line, no processing is done on the second line.

The first *unordered-list-item-processed line sequence* is therefore:

```
First item 1
```

When this *unordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a single paragraph block-element. The corresponding HTML would be:

```
<p>First item 1</p>
```

The second *unordered list item line sequence* looks like:

```
* Second item 1
Second item 2

      Code block
```

To obtain the corresponding *unordered-list-item-processed line sequence*, we need to remove the *unordered list starter string* from the beginning of the first line, and remove leading space characters, subject to a maximum of 2 space characters (because the *unordered-list-starter-string-length* is 2), from the subsequent lines.

The second *unordered-list-item-processed line sequence* is therefore:

```
Second item 1
Second item 2

    Code block
```

When this *unordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a paragraph followed by a code block. The corresponding HTML would be:

```
<p>Second item 1
Second item 2</p>

<pre><code>Code block
</code></pre>
```

The third *unordered list item line sequence* looks like:

```
* Third item 1

    * Nested item 1
```

To obtain the corresponding *unordered-list-item-processed line sequence*, we need to remove the *unordered list starter string* from the beginning of the first line, and remove leading space characters, subject to a maximum of 2 space characters, from the subsequent lines.

The third *unordered-list-item-processed line sequence* is therefore:

```
Third item 1

  * Nested item 1
```

When this *unordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a paragraph followed by an unordered list. The corresponding HTML would be:

```
<p>Third item 1</p>

<ul>
    <li>Nested item 1</li>
</ul>
```

Putting the content of all the list items together, the HTML equivalent for the complete block-element line sequence of the unordered list in this example would be:

```
<ul>
    <li><p>First item 1</p></li>
    <li><p>Second item 1
    Second item 2</p>

    <pre><code>Code block
    </code></pre></li>
    <li><p>Third item 1</p>

    <ul>
        <li>Nested item 1</li>
    </ul></li>
<ul>
```

ordered list

The block-element line sequence for an ordered list block shall have one or more lines.

The first line in the block-element line sequence would match the ordered list starter pattern (i.e. the regular expression `/^( *([0-9]+)\. +)[^ ]/`). The length of the matching substring for the first (i.e. outer) parenthesized subexpression in

the pattern is called the *ordered-list-starter-string-length*. The matching substring for the second (i.e. inner) parenthesized subexpression in the pattern is called the *ordered list starting number*.

We first divide the block-element line sequence into a series of *ordered list item line sequences*. The lines in a particular *ordered list item line sequence* correspond to one list item in the list.

Every line in the block-element line sequence that satisfies all the following conditions is called an *ordered list item start line*:

1. The line matches the ordered list starter pattern
2. The first *ordered-list-starter-string-length* characters of the line include non-space characters

Each *ordered list item start line* signifies the beginning of a new *ordered list item line sequence*. An *ordered list item line sequence* consists of the sequence of lines starting from (and inclusive of) an *ordered list item start line*, and ending at (and excluding) the next subsequent *ordered list item start line*. If there is no subsequent *ordered list item start line*, the *ordered list item line sequence* ends at the last line of the block-element line sequence.

We have now divided the block-element line sequence into a series of *ordered list item line sequences*. The first line of each *ordered list item line sequence* matches the ordered list starter pattern.

Each line in the ordered list item line sequence is processed to produce a modified sequence of lines, called the *ordered-list-item-processed line sequence*. The following processing is to be done for each line:

1. If the line is the first line of the *ordered list item line sequence*:

   The line would match the ordered list starter pattern. The matching substring for the first (i.e. outer) parenthesized subexpression in the pattern shall be removed from the beginning of the line.

2. If the line is not the first line of the *ordered list item line sequence*:

   The line would start with zero or more space characters. The leading space characters, if any, should be removed as given below:

   1. If the number of leading space characters exceeds the *ordered-list-starter-string-length*, the number of leading space characters removed should be equal to the *ordered-list-starter-string-length*.
   2. If the number of leading space characters is less than or equal to the *ordered-list-starter-string-length*, all the leading space characters should be removed.

Each ordered-list-item-processed line sequence obtained this way can be considered as the input line sequence for a sequence of block-elements nested within the list item. The result of interpreting that input line sequence further into block-elements shall form the content of the list element.

An ordered-list-item-processed line sequence has certain properties that are useful in determining how the paragraph block-elements (if any) contained within the ordered-list-item-processed line sequence should be handled.

The list elements so obtained are combined into a sequence to form the complete ordered list in the output.

The numbering for the ordered list should start from the *ordered list starting number*. For HTML output, if the *ordered list starting number* is the number '1', the corresponding `ol` start tag in the output shall not have the `start` attribute; if the *ordered list starting number* is not the number '1', the corresponding `ol` start tag in the output shall include the `start` attribute with the *ordered list starting number* as the attribute value.

For example, consider the following block-element line sequence:

```
1. First item 1

2. Second item 1
Second item 2

       Code block

3. Third item 1

    1. Nested item 1
```

When we match the first line against the ordered list starter pattern, the matching substring for the first (i.e. outer) parenthesized subexpression is obtained as `1. ` (i.e. the number '1', followed by a dot, followed by a single space character). The *ordered-list-starter-string-length* is therefore 3. Also, the *ordered list starting number* is identified as the number '1'.

The 1st, 3rd, 8th and 10th lines in the block-element line sequence match the *ordered list starter pattern*, but only the 1st, 3rd and 8th lines are such that the first 3 characters of the line include non-space characters. So only the 1st, 3rd and 8th lines

are are *ordered list item start lines*. Therefore, there are three *ordered list item line sequences* in the above example, as follows:

1. The lines 1 and 2 form the first *ordered list item line sequence*
2. The lines 3-7 form the second *ordered list item line sequence*
3. The lines 8-10 form the third and last *ordered list item line sequence*

Each *ordered list item line sequence* is then processed to obtain the *ordered-list-item-processed line sequence*. When we treat each *ordered-list-item-processed line sequence* as an input line sequence, we can recognize nested block elements in it.

The first *ordered list item line sequence* looks like:

```
1. First item 1
```

To obtain the corresponding *ordered-list-item-processed line sequence*, we need to match the first line against the ordered list starter pattern and remove the matching substring for the first parenthesized subexpression. The matching substring in this case is `1. ` (i.e. the number '1', followed by a dot, followed by a single space character). Since the second line is a blank line, no processing is done on the second line.

The first *ordered-list-item-processed line sequence* is therefore:

```
First item 1
```

When this *ordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a single paragraph block-element. The corresponding HTML would be:

```
<p>First item 1</p>
```

The second *ordered list item line sequence* looks like:

```
2. Second item 1
Second item 2

       Code block
```

To obtain the corresponding *ordered-list-item-processed line sequence*, we need to match the first line against the *ordered list starter pattern* and remove the matching substring for the first parenthesized subexpression. The matching substring in this case is `2. ` (i.e. the number '2', followed by a dot, followed by a single space character). From subsequent lines, we need to remove leading space characters, subject to a maximum of 3 space characters (because the *ordered-list-starter-string-length* is 3).

The second *ordered-list-item-processed line sequence* is therefore:

```
Second item 1
Second item 2

    Code block
```

When this *ordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a paragraph followed by a code block. The corresponding HTML would be:

```
<p>Second item 1
Second item 2</p>

<pre><code>Code block
</code></pre>
```

The third *ordered list item line sequence* looks like:

```
3. Third item 1

    1. Nested item 1
```

To obtain the corresponding *ordered-list-item-processed line sequence*, we need to match the first line against the ordered list starter pattern and remove the matching substring for the first parenthesized subexpression. The matching substring in this case is `3. ` (i.e. the number '3', followed by a dot, followed by a single space character). From subsequent lines, we

need to remove leading space characters, subject to a maximum of 3 space characters (because the *ordered-list-starter-string-length* is 3).

The third *ordered-list-item-processed line sequence* is therefore:

```
Third item 1

 1. Nested item 1
```

When this *ordered-list-item-processed line sequence* is processed as an input line sequence to identify block-elements in it, we get a paragraph followed by an ordered list. The corresponding HTML would be:

```
<p>Third item 1</p>

<ol>
    <li>Nested item 1</li>
</ol>
```

Putting the content of all the list items together, the HTML equivalent for the complete block-element line sequence of the ordered list in this example would be:

```
<ol>
    <li><p>First item 1</p></li>
    <li><p>Second item 1
    Second item 2</p>

    <pre><code>Code block
    </code></pre></li>
    <li><p>Third item 1</p>

    <ol>
        <li>Nested item 1</li>
    </ol></li>
</ol>
```

paragraph

The block-element line sequence for a paragraph block shall have one or more lines.

The lines in the block-element line sequence are joined together into a single sequence of characters, with a line break after each line. The resulting sequence of characters is trimmed to give the *paragraph text*. The result of interpreting the *paragraph text* as a text span sequence shall form the content of the paragraph element.

For HTML output, the *paragraph text* needs to be run through a HTML parser to determine how the content of the paragraph element should be presented.

For HTML output, the content of the paragraph element shall be enclosed in `p` tags, unless any of the following conditions is satisfied:

1. The *paragraph text* contains an unmatched HTML tag (i.e. open tag without a close tag, or a close tag without an open tag)
2. The *paragraph text* contains a misnested HTML tag (i.e. close tag at the wrong position)
3. The *paragraph text* contains a HTML element that is not a phrasing-html-element
4. The *paragraph text* contains a HTML comment
5. The block-element line sequence for the paragraph block is the first block-element line sequence of its parent line sequence, and the parent line sequence is a top-packed list-item-processed line sequence
6. The block-element line sequence for the paragraph block is the last block-element line sequence of its parent line sequence, but not the second block-element line sequence of its parent line sequence, and the parent line sequence is a bottom-packed list-item-processed line sequence

If one or more of the above 6 conditions is satisfied, the HTML output of the paragraph shall be the same as the content of the paragraph, without wrapping it in `p` tags.

Properties of list item line sequences

A *list item line sequence* denotes either an unordered list item line sequence or an ordered list item line sequence.

A *list-item-processed line sequence* denotes either an unordered-list-item-processed line sequence or an ordered-list-item-processed line sequence.

Top-packed list item line sequences

A list item line sequence, *S*, is said to be *top-packed* if, and only if, *S* satisfies any of the following conditions:

1. *S* is the only list item line sequence in the block-element line sequence

   (or)

2. *S* is the first list item line sequence in the block-element line sequence, and the last line of *S* is not a blank line

   (or)

3. *S* is not the first list item line sequence in the block-element line sequence, and the line immediately before the first line of *S* in the block-element line sequence is not a blank line

If a list item line sequence is *top-packed*, the list-item-processed line sequence obtained from it is also said to be *top-packed*. Otherwise, the list-item-processed line sequence is not said to be *top-packed*.

Bottom-packed list item line sequences

A list item line sequence, *S*, is said to be *bottom-packed* if, and only if, *S* satisfies any of the following conditions:

1. *S* is the only list item line sequence in the block-element line sequence

   (or)

2. *S* is the last list item line sequence in the block-element line sequence, and the line immediately before the first line of *S* in the block-element line sequence is not a blank line

   (or)

3. *S* is not the last list item line sequence in the block-element line sequence, and the last line of *S* is not a blank line

If a list item line sequence is *bottom-packed*, the list-item-processed line sequence obtained from it is also said to be *bottom-packed*. Otherwise, the list-item-processed line sequence is not said to be *bottom-packed*.

Identifying span-elements

A **text span sequence** is a sequence of span-level vfmd constructs in a paragraph or header block.

To interpret a non-empty sequence of characters, called the **input character sequence**, as a *text span sequence*, we need to identify the type and extent of the span-elements in the input.

Some characters in the *input character sequence* form the text content, and the other characters denote how the text content is to be "marked up". The characters that denote "mark up" form *span tags*, and the other characters form *text fragments*.

There are three kinds of *span tags*: *opening span tags*, *closing span tags* and *self-contained span tags*.

For example, consider the following *input character sequence*:

```
The **`ls` command** [_lists_ files](/ls-cmd).
```

The above *input character sequence* can be broken down into the following:

```
"The "       : text fragment
"**"         : opening span tag (emphasis)
"`ls`"       : self-contained span tag (code)
" command"   : text fragment
"**"         : closing span tag (emphasis)
"["          : opening span tag (link)
"_"          : opening span tag (emphasis)
"lists "     : text fragment
"_"          : closing span tag (emphasis)
"files"      : text fragment
"](/ls-cmd)" : closing span tag (link)
"."          : text fragment
```

To identify and interpret the *span tags* in the *input character sequence*, we make use of a **stack of potential opening span tags**. Each node in the *stack of potential opening span tags* eventually might or might not get interpreted as an *opening span tag*. If a corresponding *closing span tag* is identified, the node gets interpreted as an *opening span tag*; else, it gets identified as a *text fragment*.

Initially, the *stack of potential opening span tags* is empty. The stack grows upwards: the bottommost node in the stack is the first one added to the stack, and pushing a node onto the stack places it on top of the stack.

Each node in the stack contains the following properties:

1. A *tag string* that contains the substring of the *input character sequence* that forms the *span tag*
2. A *node type* that indicates what types of span element this node might be opening, the possible values being: *asterisk emphasis node*, *underscore emphasis node*, *link node* or *raw html node*
3. A *linked content start position* that is set only if the *node type* is equal to *link node*
4. A *html tag name* that is set only if the *node type* is equal to *raw html node*

The topmost node in the stack of potential opening span tags is called the *top node*.

A node *n* is said to be the *topmost node* of type *t* if, and only if, all the following conditions are satisfied:

1. The *node type* of *n* is equal to *t*
2. The node *n* is present in the stack of potential opening span tags
3. There is no other node *m* (where *n* != *m*) such that both the following are true:
    1. The node *m* is above the node *n* in the stack of potential opening span tags
    2. The *node type* of *m* is equal to *t*, or the *node type* of *m* is equal to *raw html node*

The *topmost node* of type *t* is said to be *null* if, and only if, any of the following conditions is true:

1. The stack of potential opening span tags does not contain any node whose *node type* is equal to *t*

    (or)

2. All nodes whose *node type* is equal to *t* in the stack of potential opening span tags have a *html node* above them (where *html node* means a node whose *node type* is *raw html node*)

To identify and interpret the *span tags* in the input character sequence, the procedure for identifying span tags shall be used.

Procedure for identifying span tags

In this section, we discuss the procedure to identify and interpret the *span tags* in the input character sequence.

The procedure involves iterating over the characters in the input character sequence. The current character position in the input character sequence is called the **current-position**. When *current-position* is 0, the first character in the input character sequence is said to be the character at the *current-position*; when *current-position* is 1, the second character in the input character sequence is said to be the character at the *current-position*, and so on. The substring of the input character sequence starting from and including the character at the *current-position* and ending at the end of the input character sequence is called the **remaining-character-sequence**. When *current-position* is 0, the *remaining-character-sequence* is equal to the input character sequence.

The root procedure in turn invokes other procedures. In the method described here, global variables are used to communicate information from invoked procedures back to the root procedure, but better means can be adopted by an implementation. The global variable used is:

1. *consumed-character-count*: The number of characters that were consumed to form a *span tag* or a *text fragment*

The procedure to identify and interpret the *span tags* is as follows:

1. Set current-position as 0
2. Set *consumed-character-count* as 0
3. Depending on the character at the current-position, invoke the appropriate procedure, as given below:
    1. If the character at the current-position is either an unescaped `[` (open square bracket) character, or an unescaped `]` (close square bracket) character, invoke the procedure for identifying link tags
    2. If the character at the current-position is either an unescaped `*` (asterisk) character, or an unescaped `_` (underscore or low line) character, then invoke the procedure for identifying emphasis tags
    3. If the character at the current-position is an unescaped `` ` `` (backtick) character, then invoke the procedure for identifying code-span tags
    4. If the character at the current-position is an unescaped `!` (exclamation mark) character, and if the remaining-character-sequence matches the regular expression pattern `/!\[/`, then invoke the procedure for identifying image tags

5. If *consumed-character-count* is equal to 0, then invoke the procedure for detecting automatic links

6. If *consumed-character-count* is equal to 0, and if the character at the current-position is an unescaped `<` (left angle bracket) character, then invoke the procedure for identifying HTML tags

7. If *consumed-character-count* is equal to 0, interpret the character at the current-position to be part of a *text fragment*, and set *consumed-character-count* as 1

4. Increment current-position by *consumed-character-count*

5. If current-position is less than the length of the input character sequence, go to Step 2

6. All nodes in the stack of potential opening span tags with *node type* not equal to *raw html node* should be interpreted as *text fragments*

The *text fragments* identified in the above procedure should be handled as specified in processing text fragments.

An implementation can extend the core vfmd syntax to support additional syntax elements. For every additional span-level syntax element, the implementation shall insert a rule in the above rule list, at a position at which it makes sense to recognize the particular construct. The rule shall determine whether a span-level construct begins at a particular position in the input character sequence or not, and if it does, how it should be interpreted and output.

Procedure for identifying link tags

This procedure assumes that the character at the current-position is either an unescaped `[` character or an unescaped `]` character.

If the character at the current-position is a `[` character, it implies that the `[` can potentially get interpreted as an *opening link tag* in the future. In this case, the following is done:

1. A new node is pushed onto the stack of potential opening span tags with the following properties:

    1. The *tag string* of the node is set to the `[` character at the current position

    2. The *node type* of the node is set as *link node*

    3. The *linked content start position* of the node is set to (current-position + 1)

2. Set consumed-character-count to 1

If the character at the current-position is a `]` character, it might be the start of a *closing link tag*. In this case, the following is done:

1. If the topmost node of type *link node* is not *null*, and if the remaining-character-sequence matches the regular expression pattern `/^\]\s*\[(([^\\[\]\`]|\\.)+)\]/` (Example: `] [ref id]`), then the following is done:

    1. The matching substring for the whole of the pattern is identified as a **closing link tag**.

    2. The matching substring for the first (i.e. outer) parenthesized subexpression in the pattern is simplified to obtain the *reference id string*

    3. If the topmost node of type *link node* is not already the top node, all nodes above it are popped off and interpreted as *text fragments*

    4. The top node is interpreted as an *opening span tag*, or more specifically, as an **opening link tag**

    5. The *closing link tag* is said to correspond to the *opening link tag*, and any *span tags* or *text fragments* occuring between the *opening link tag* and the *closing link tag* are considered to form the *enclosed content* of the link

    6. The *reference id string* shall be used to look up the actual link url and link title from the link reference association map.

        If the link reference association map contains an entry for *reference id string*, then the output shall have the *enclosed content* linked to the link url and link title specified in the entry for the *reference id string* in the link reference association map. For HTML output, the link url should be URL-escaped; the link title should be de-escaped and then attribute-value-escaped.

        If the link reference association map does not contain an entry for *reference id string*, then the output shall have the *enclosed content* without being part of a link, enclosed within the text forming the *opening link tag* and the text forming the *closing link tag*. For HTML output, the text forming the *closing link tag* should be de-escaped and then html-text-escaped before being output.

    7. The top node is popped off

8. All nodes with *node type* equal to *link node* are removed from the stack of potential opening span tags and interpreted as *text fragments*

9. Set consumed-character-count to the number of characters in the *closing link tag*

2. If the topmost node of type *link node* is not *null*, and if both the following conditions are satisfied:

    1. The remaining-character-sequence matches one of the following regular expression patterns:

        1. URL without angle brackets: `/^\]\s*\(\s*([^\(\)<>\`\s]+)([\)\s].*)$/`

            Example: `] (http://www.example.net` + *residual-link-attribute-sequence*

        2. URL within angle brackets: `/^\]\s*\(\s*<([^<>\`]*)>([\)\s].*)$/`

            Examples:
            `](<http://example.net>` + *residual-link-attribute-sequence*
            `] ( <http://example.net/?q=)>` + *residual-link-attribute-sequence*

        In case of either pattern, the matching substring for the first parenthesised subexpression shall be called the *unprocessed url string*, and the matching substring for the second parenthesized subexpression shall be called the *residual-link-attribute-sequence*. Any whitespace characters in the *unprocessed url string* are removed, and the resultant string is called the *link url string*.

        The position at which the *residual-link-attribute-sequence* starts within the remaining-character-sequence (i.e. the number of characters present in the remaining-character-sequence before the start of the *residual-link-attribute-sequence*) shall be called the *url-pattern-match-length*.

    2. The *residual-link-attribute-sequence* matches one of the following regular expression patterns:

        1. Just the closing paranthesis: `/^\s*\)/`

            Example: `)`

            If this is the matching pattern, the *title string* is said to be *null*.

        2. Title and closing paranthesis:
            `/^\s*(″(([^\\″\`]|\\.)*)″|’(([^\\’\`]|\\.)*)’)\s*\)/`

            Examples:
            `″Title″)`
            `’Title’)`
            `″A (nice) \″title\″ for the ’link’″)`

            If this is the matching pattern, the matching substring for the first (i.e. outer) parenthesized subexpression in the pattern is called the *attributes-string*. The *attributes-string* will be a quoted string, and the enclosed string of the quoted string is called the *unprocessed title string*. Any line break characters in the *unprocessed title string* are removed, and the resultant string is called the *title string*.

        The number of characters in the *residual-link-attribute-sequence* that were consumed in matching the whole of the matching pattern is called the *attributes-pattern-match-length*.

    then, the following is done:

    1. Let *close-link-tag-length* be equal to the sum of the *url-pattern-match-length* and the *attributes-pattern-match-length*. The first *close-link-tag-length* characters of the remaining-character-sequence are collectively identified as a **closing link tag**

    2. If the topmost node of type *link node* is not already the top node, all nodes above it are popped off and interpreted as *text fragments*

    3. The top node (which should have its *node type* equal to *link node*) is interpreted as an *opening span tag*, or more specifically, as an **opening link tag**

    4. The *closing link tag* is said to correspond to the *opening link tag*, and any *span tags* or *text fragments* occuring between the *opening link tag* and the *closing link tag* are considered to form the *enclosed content* of the link

    5. The *link url string* shall be used as the link url for the link. If the *title string* is not *null*, the *title string* shall be used as the title for the link. The output shall have the *enclosed content* linked to the link url and link title. For HTML output, the link url should be URL-escaped; the link title should be de-escaped and then attribute-value-escaped.

    6. The top node is popped off

7. All nodes with *node type* equal to *link node* are removed from the stack of potential opening span tags and interpreted as *text fragments*

8. Set consumed-character-count to *close-link-tag-length*

3. If neither of the above conditions are satisfied, and if the topmost node of type *link node* is not *null*, and if the character at the current-position is a `]` character, then the following is done:

    1. Let *empty-ref-pattern* be the regular expression pattern `/^(\]\s*\[\s*\])/` (Example: `][]`)

        If the remaining-character-sequence matches the *empty-ref-pattern*, then the length of the matching substring for the whole pattern is said to be the *close-link-tag-length*.

        If the remaining-character-sequence does not match the *empty-ref-pattern*, then the *close-link-tag-length* is said to be 1.

        The first *close-link-tag-length* characters of the remaining-character-sequence are collectively identified as a **closing link tag**

    2. If the topmost node of type *link node* is not already the top node, all nodes above it are popped off and interpreted as *text fragments*

    3. The top node is interpreted as an *opening span tag*, or more specifically, as an **opening link tag**

    4. The *closing link tag* is said to correspond to the *opening link tag*, and any *span tags* or *text fragments* occuring between the *opening link tag* and the *closing link tag* are considered to form the *enclosed content* of the link

    5. Let *reference id start position* be the *linked content start position* of the *top node*; Let *reference id end position* be (current-position - 1). The substring of the *input character sequence* starting from the *reference id start position* and ending at the *reference id end position*, both inclusive, is simplified to obtain the *reference id string*.

        The *reference id string* shall be used to look up the actual link url and link title from the link reference association map.

        If the link reference association map contains an entry for *reference id string*, then the output shall have the *enclosed content* linked to the link url and link title specified in the entry for the *reference id string* in the link reference association map. For HTML output, the link url should be URL-escaped; the link title should be de-escaped and then attribute-value-escaped.

        If the link reference association map does not contain an entry for *reference id string*, then the output shall have the *enclosed content* without being part of a link, enclosed within the text forming the *opening link tag* and the text forming the *closing link tag*. For HTML output, the text forming the *closing link tag* should be de-escaped and then html-text-escaped before being output.

    6. The top node is popped off

    7. All nodes with *node type* equal to *link node* are removed from the stack of potential opening span tags

    8. Set consumed-character-count to the number of characters in the *closing link tag*

4. If the topmost node of type *link node* is *null*, and if the character at the current-position is a `]` character, then the `]` at the current-position is interpreted as a *text fragment*, and consumed-character-count is set to 1

Procedure for identifying emphasis tags

This procedure assumes that the character at the current-position is either an unescaped `*` character or an unescaped `_` character.

We define **emphasis-fringe-rank** of a character based on the 'General_Category' unicode property as follows:

1. If the 'General_Category' unicode property of the character is one of the following: Zs, Zl, Zp, Cc or Cf, then the *emphasis fringe rank* of the character is said to be 0.

    For example, space and line break characters have an *emphasis fringe rank* of 0.

2. If the 'General_Category' unicode property of the character is one of the following: Pc, Pd, Ps, Pe, Pi, Pf, Po, Sc, Sk, Sm or So, then the *emphasis fringe rank* of the character is said to be 1.

    For example, the following characters have an *emphasis fringe rank* of 1: `,`, `.`, `(`, `)`, `+`, `>`

3. If the 'General_Category' unicode property of the character is not one of the following: Zs, Zl, Zp, Cc, Cf, Pc, Pd, Ps, Pe, Pi, Pf or Po, then the *emphasis fringe rank* of the character is said to be 2.

    For example, alphanumeric characters have an *emphasis fringe rank* of 2.

Given that the character at the current-position is either `*` or `_`, then the remaining-character-sequence will definitely match one of the following regular expression patterns:

1. At the end of the input character sequence: `/^([\*_]+)$/`
2. In the middle of the input character sequence: `/^([\*_]+)([^\*_])/`

In case of either pattern, the matching substring for the first parenthesized subexpression is called the *emphasis indicator string*.

In case the match is with the first regular expression pattern given above, then the *right-fringe-rank* of the emphasis indicator string is said to be 0. In case the match is with the second regular expression pattern given above, then the *right-fringe-rank* of the emphasis indicator string is the emphasis-fringe-rank of the single character in the matching substring for the second parenthesized subexpression in the pattern.

If the current-position is equal to 0, the *left-fringe-rank* of the emphasis indicator string is said to be 0. If the current-position is greater than 1, then the *left-fringe-rank* of the emphasis indicator string is the emphasis-fringe-rank of the character at the previous position (i.e. at current-position minus 1).

For a given emphasis indicator string, if its left-fringe-rank is lesser than its right-fringe-rank, the emphasis indicator string is said to be *left-flanking*. On the other hand, if its right-fringe-rank is lesser than its left-fringe-rank, the emphasis indicator string is said to be *right-flanking*. If the left-fringe-rank of an emphasis indicator string is equal to its right-fringe-rank, the emphasis indicator string is said to be *non-flanking*.

Consider the following example:

```
***Shaken*, ** not _stirred_**.
```

There are 5 *emphasis indicator strings* in the above example. The *left-fringe-rank* and *right-fringe-rank* of each is given below:

#	*emphasis indicator string*	*left fringe rank*	*right fringe rank*	flankingness
1	`***`	0	2	*left-flanking*
2	`*`	2	1	*right-flanking*
3	`**`	0	0	*non-flanking*
4	`_`	0	2	*left-flanking*
5	`_**`	2	1	*right-flanking*

An emphasis indicator string can contain both `*` and `_` characters. When we split the emphasis indicator string into substrings composed of the same character, with no adjacent substring having a common character, we get a list of *emphasis tag strings*.

For example:

emphasis indicator string	List of *emphasis tag strings*
`***`	[`***`]
`***___**`	[`***` , `___` , `**`]
`__*_**__`	[`__` , `*` , `_` , `**` , `__`]

An emphasis tag string shall either be composed entirely of `*` characters, or be composed entirely of `_` characters. If it's composed entirely of `*` characters, the *constituent character* of the emphasis tag string is said to be `*`. If it's composed entirely of `_` characters, the *constituent character* of the emphasis tag string is said to be `_`.

If the emphasis indicator string is non-flanking, then the emphasis indicator string is interpreted as a *text fragment*. The consumed-character-count is set to the length of the emphasis indicator string.

If the emphasis indicator string is left-flanking, then the emphasis tag strings in the emphasis indicator string can potentially become *opening emphasis tags*. In this case, the following shall be done:

1. For each emphasis tag string in the emphasis indicator string (listed in the order in which the emphasis tag string appears in the emphasis indicator string) a new node is pushed onto the stack of potential opening span tags with the following properties:
    1. The *tag string* of the node is set to the *emphasis tag string*

2. If the constituent character of the emphasis tag string is `*`, then the *node type* of the node is set as *asterisk emphasis node*; if the constituent character of the *emphasis tag string* is `_`, then the *node type* of the node is set as *underscore emphasis node*

2. Set consumed-character-count to the length of the emphasis indicator string

If the emphasis indicator string is right-flanking, then the emphasis tag strings in the emphasis indicator string can potentially be interpreted as *closing emphasis tags*. In this case, the following shall be done:

1. Set *current-tag-string* to the first emphasis tag string in the emphasis indicator string

2. If the constituent character of the *current-tag-string* is `*`, then the *matching emphasis node* is the topmost node of type *asterisk emphasis node*; if the constituent character of the *current-tag-string* is `_`, then the *matching emphasis node* is the topmost node of type *underscore emphasis node*

3. If *matching emphasis node* is *null*, then the *emphasis tag string* is interpreted as a *text fragment*

4. If the *matching emphasis node* is not *null*, and if it is not already the top node, then all nodes above it are popped off and interpreted as *text fragments*

5. If the *matching emphasis node* is not *null*, then invoke the procedure for matching emphasis tag strings. The *current-tag-string* and/or the top node can get modified within that procedure.

6. If the *current-tag-string* is empty, and if there are any more unprocessed emphasis tag strings in the emphasis indicator string, set the *current-tag-string* to the next emphasis tag string

7. If the *current-tag-string* is not empty, go to Step 2

8. Set consumed-character-count to the length of the emphasis indicator string

Procedure for matching emphasis tag strings

This procedure describes how the *current-tag-string* is to be matched with the *matching emphasis node* at the top of the stack of potential opening span tags.

In this procedure, the top node is assumed to be of a *node type* that matches the constituent character of the *current-tag-string*. If the constituent character of the *current-tag-string* is `*`, the *node type* of the top node should be *asterisk emphasis node*. If the constituent character of the *current-tag-string* is `_`, the *node type* of the top node should be *underscore emphasis node*.

Let the *tag string* of the top node be called the *top node tag string*. The *top node tag string* is compared with the *current-tag-string*.

1. If the *top node tag string* and the *current-tag-string* are exactly the same strings, then:

    1. The top node is interpreted as an **opening emphasis tag**

    2. The *current-tag-string* is interpreted as **closing emphasis tag**

    3. The *closing emphasis tag* is said to correspond to the *opening emphasis tag*, and any *span tags* or *text fragments* occuring between the *opening emphasis tag* and the *closing emphasis tag* are considered to constitute the emphasized content

    4. Set the *current-tag-string* to *null*

    5. The top node is popped off

2. If the *top node tag string* and the *current-tag-string* are not exactly the same strings, and if the *current-tag-string* is a substring of the *top node tag string*, then:

    1. Let the length of the *current-tag-string* be called the *current-tag-string-length*.

        The last *current-tag-string-length* characters of the *top node tag string* are interpreted as an **opening emphasis tag**.

    2. The whole of the *current-tag-string* is interpreted a **closing emphasis tag**

    3. The *closing emphasis tag* is said to correspond to the *opening emphasis tag*, and any *span tags* or *text fragments* occuring between the *opening emphasis tag* and the *closing emphasis tag* are considered to constitute the emphasized content

    4. Set the *current-tag-string* to *null*

    5. The characters of the *top node tag string* that were interpreted as the *opening emphasis tag* are removed from the *tag string* of the top node while the top node is still retained in the stack of potential opening span tags

3. If the *top node tag string* and the *current-tag-string* are not exactly the same strings, and if the *top node tag string* is a substring of the *current-tag-string*, then:

    1. The top node is interpreted as an *opening span tag*, or more specifically, as an **opening emphasis tag**
    2. Let the length of the *top node tag string* be called the *top-node-tag-string-length*.

        The first *top-node-tag-string-length* characters of the *current-tag-string* interpreted as a *closing span tag*, or more specifically, as a **closing emphasis tag**.
    3. The *closing emphasis tag* is said to correspond to the *opening emphasis tag*, and any *span tags* or *text fragments* occuring between the *opening emphasis tag* and the *closing emphasis tag* are considered to constitute the emphasized content
    4. The characters of the *current-tag-string* that were interpreted as the *closing emphasis tag* are removed from the *current-tag-string*
    5. The top node is popped off

4. If any of the above conditions are satisfied, then we would have identified an *opening emphasis tag* and a *closing emphasis tag*. The length of the *opening emphasis tag* and the *closing emphasis tag* should be the same. Let the length of the *opening emphasis tag* or the *closing emphasis tag* be called the *emphasis-tag-length*.

    The kind of emphasis that is to be applied is determined as follows:

    1. If the *emphasis-tag-length* is 1, use an emphasis of kind *emphatic stress*. In HTML output, this shall be output as an `em` element.
    2. If the *emphasis-tag-length* is 2, use an emphasis of kind *strong importance*. In HTML output, this shall be output as a `strong` element.
    3. If the *emphasis-tag-length* is 3 or above, use both an emphasis of kind *strong importance* and an emphasis of kind *emphatic stress*. In HTML output, this shall be output as an `em` element nested within a `strong` element.

Procedure for identifying code-span tags

This procedure assumes that the character at the current-position is an unescaped `` ` `` character.

1. The remaining-character-sequence shall match one of the following regular expression patterns:

    1. Backticks followed by a non-backtick: `/^(`+)([^`].*)$/`

        Example: ```` ```p ````
    2. Backticks at the end of the *input character sequence*: `/^(`+)$/`

        Example: ```` ``` ````

    In case of either pattern, the matching substring for the first parenthesized subexpression in the pattern is called the *opening-backticks-string*. The length of the *opening-backticks-string* is called the *opening-backticks-count*.

    In case the match is with the first regular expression pattern, the matching substring for the second parenthesized subexpression in the pattern shall be called the *residual-code-span-sequence*. In case the match is with the second regular expression pattern, the *residual-code-span-sequence* is said to be *null*.

2. Set *code-content-length* to 0
3. If the *residual-code-span-sequence* matches one of the following regular expression patterns:

    1. Non-backticks followed by backticks followed by a non-backtick: `/^([^`]+)(`+)([^`].*)$/`

        Example: ```` printf()```. ````
    2. Non-backticks followed by backticks, at the end of the *input character sequence*: `/^([^`]+)(`+)$/`

        Example: ```` printf()``` ````

    then, the following is done:

    1. The matching substring for the first parenthesized subexpression in the matching pattern is called the *code-fragment-string*. The matching substring for the second parenthesized subexpression in the matching pattern is called the *backticks-fragment-string*.
    2. The *code-content-length* is incremented by the length of the *code-fragment-string*

3. In case the match is with the first regular expression pattern, the *residual-code-span-sequence* is set to the matching substring for the third parenthesized subexpression in the pattern. In case the match is with the second regular expression pattern, the *residual-code-span-sequence* is set to *null*.

4. If the length of the *backticks-fragment-string* is equal to the length of the *opening-backticks-count*, then the *backticks-fragment-string* is identified as the *closing-code-string*

5. If the length of the *backticks-fragment-string* is not equal to the *opening-backticks-count*, then the *code-content-length* is incremented by the length of the *backticks-fragment-string*

4. If a *closing-code-string* has not yet been identified, and if the *residual-code-span-sequence* contains one or more `` ` `` characters, go to Step 3

5. If a *closing-code-string* has not yet been identified, the *opening-backticks-string* is identified as a *text fragment*, and consumed-character-count is set to the length of the *opening-backticks-string*

6. If a *closing-code-string* has been identified, the following is done:

    1. Let *code-span-length* be equal to ((*opening-backticks-count* * 2) + *code-content-length*)

    2. The first *code-span-length* characters of the remaining-character-sequence is identified as a **code span tag**

    3. Among the characters that form the *code span tag*, the first *opening-backticks-count* characters and the last *opening-backticks-count* characters are considered to be markup. The middle *code-content-length* characters constitute the *unprocessed-code-content-string*. The *unprocessed-code-content-string* is trimmed to form the content of the code span. For HTML output, the content of the code-span should be html-code-escaped.

    4. Set consumed-character-count to *code-span-length*

Procedure for identifying image tags

This procedure assumes that the character at the current-position is an unescaped `!` character, and that the immediate next character is a `[` character.

The regular expression pattern `/^!\[(([^\\\[\]\`]|\\.)*)(\].*)$/` is called the **image-tag-starter-pattern**.

Example: `![alt text` + *residual-image-sequence*

If the remaining-character-sequence does not match the image-tag-starter-pattern, then the first 2 characters of the remaining-character-sequence (which should form the string `![`) are interpreted as a *text fragment*.

If the remaining-character-sequence matches the image-tag-starter-pattern, then:

1. The matching substring for the first parenthesized subexpression in the pattern is called the *image-alt-text-string*

2. The matching substring for the last parenthesized subexpression in the pattern is called the *residual-image-sequence*

3. The position at which the *residual-image-sequence* starts within the remaining-character-sequence (i.e. the number of characters present in the remaining-character-sequence before the start of the *residual-image-sequence*) shall be called the *alt-text-pattern-match-length*

If the remaining-character-sequence matches the image-tag-starter-pattern, then the following is done:

1. If the residual-image-sequence matches the regular expression pattern `/^\]\s*\[(([^\\\[\]\`]|\\.)*)\]/` (Example: `] [ref id]`), then the following is done:

    1. The matching substring for the first parenthesized subexpression in the pattern is simplified to obtain the *reference id string*

    2. The length of the matching substring for the whole of the pattern is called the *image-ref-close-sequence-length*.

        Let *image-ref-tag-length* be equal to the sum of the alt-text-pattern-match-length and the *image-ref-close-sequence-length*. The first *image-ref-tag-length* characters of the remaining-character-sequence are collectively identified as an **image tag**.

    3. The *reference id string* shall be used to look up the actual image url and image title from the link reference association map.

        If the link reference association map contains an entry for *reference id string*, then the output shall have the source of the image as the link url and the title of the image as the link title (if available) specified in the entry for the *reference id string* in the link reference association map. The image-alt-text-string shall be used as the alternate text for the image. For HTML output, the link url of the image should be URL-escaped; the title of the image and the alternate text for the image should be de-escaped and then attribute-value-escaped.

If the link reference association map does not contain an entry for *reference id string*, then the output shall not include an image for this *image tag*. Instead, the first *image-ref-tag-length* characters of the remaining-character-sequence are output as text. For HTML output, this text should be html-text-escaped.

4. Set consumed-character-count to *image-ref-tag-length*

2. If the residual-image-sequence matches the regular expression pattern `/^\]\s*\(/` and if both the following conditions are satisfied:

    1. The residual-image-sequence matches one of the following regular expression patterns:

        1. URL without angle brackets: `/^\]\s*\(\s*([^\(\)<>\`\s]+)([\)\s].*)$/`

            Example: `] (http://www.example.net/image.jpg` + *residual-image-attribute-sequence*

        2. URL within angle brackets: `/^\]\s*\(\s*<([^<>\`]*)>([\)].+)$/`

            Examples:
            `](<http://example.net/image.jpg>` + *residual-image-attribute-sequence*
            `] ( <http://example.net/image(1).jpg>` + *residual-image-attribute-sequence*

        In case of either pattern, the matching substring for the first parenthesised subexpression shall be called the *unprocessed image source string*, and the matching substring for the second parenthesized subexpression shall be called the *residual-image-attribute-sequence*. Any whitespace characters in the *unprocessed image source string* are removed, and the resultant string is called the *image url string*.

        The position at which the *residual-image-attribute-sequence* starts within the *residual-image-sequence* (i.e. the number of characters present in the *residual-image-sequence* before the start of the *residual-image-attribute-sequence*) shall be called the *image-source-pattern-match-length*.

    2. The *residual-image-attribute-sequence* matches one of the following regular expression patterns:

        1. Just the closing paranthesis: `/^\s*\)/`

            Example: `)`

            If this is the matching pattern, the *title string* is said to be *null*.

        2. Title and closing paranthesis:
            `/^\s*("(([^"\\\`]|\\.)*)"|'(([^'\\\`]|\\.)*)')\s*\)/`

            Examples:
            `"Title")`
            `'Title')`
            `"A (nice) \"title\" for the 'image'")`

            If this is the matching pattern, the matching substring for the first (i.e. outer) parenthesized subexpression in the pattern is called the *attributes-string*. The *attributes-string* will be a quoted string, and the enclosed string of the quoted string is called the *unprocessed title string*. Any line break characters in the *unprocessed title string* are removed, and the resultant string is called the *title string*.

        The number of characters in the *residual-image-attribute-sequence* that were consumed in matching the whole of the matching pattern is called the *image-attributes-pattern-match-length*.

    then, the following is done:

    1. Let *image-src-tag-length* be equal to the sum of alt-text-pattern-match-length and *image-source-pattern-match-length* and *image-attributes-pattern-match-length*. The first *image-src-tag-length* characters of the remaining-character-sequence are collectively identified as an **image tag**.

    2. The *image url string* shall be used as the source of the image. If *title string* is not null, the *title string* shall be used as the title of the image. The image-alt-text-string shall be used as the alternate text for the image. For HTML output, the link url of the image should be URL-escaped; the title of the image and the alternate text for the image should be de-escaped and then attribute-value-escaped.

    3. Set consumed-character-count to *image-src-tag-length*

3. If neither of the above conditions are satisfied, then the following is done:

    1. Let *empty-ref-pattern* be the regular expression pattern `/^(\]\s*\[\s*\])/` (Example: `][]`)

        If the residual-image-sequence matches the *empty-ref-pattern*, then the length of the matching substring for the whole pattern is said to be the *image-ref-close-sequence-length*.

If the residual-image-sequence does not match the *empty-ref-pattern*, then the *image-ref-close-sequence-length* is said to be 1.

2. Let *image-ref-tag-length* be equal to the sum of alt-text-pattern-match-length and *image-ref-close-sequence-length*. The first *image-ref-tag-length* characters of the remaining-character-sequence are collectively identified as an **image tag**.

3. Let *reference id string* be the string obtained on simplifying the image-alt-text-string.

   The *reference id string* shall be used to look up the actual image url and image title from the link reference association map.

   If the link reference association map contains an entry for *reference id string*, then the output shall have the source of the image as the link url and the title of the image as the link title (if available) specified in the entry for the *reference id string* in the link reference association map. The image-alt-text-string shall be used as the alternate text for the image. For HTML output, the link url of the image should be URL-escaped; the title of the image and the alternate text for the image should be de-escaped and then attribute-value-escaped.

   If the link reference association map does not contain an entry for *reference id string*, then the output shall not include an image for this *image tag*. Instead, the first *image-ref-tag-length* characters of the remaining-character-sequence are output as text. For HTML output, this text should be html-text-escaped.

4. Set consumed-character-count to *image-ref-tag-length*

Procedure for detecting automatic links

We define a **word-separator** character to be a unicode code point whose 'General_Category' unicode property has one of the following values:

1. One of: Zs, Zl, Zp (i.e. a 'Separator')

   (or)

2. One of: Pc, Pd, Ps, Pe, Pi, Pf, Po (i.e. a 'Punctuation')

   (or)

3. One of: Cc, Cf

For example, the space character, the line break character, `.`, `,`, `(`, `)` are all *word-separator* characters.

We define an **speculative-url-end** character to be a word-separator character that is not a U+002F (SLASH) character.

If any one of the following conditions are satisfied:

1. The character at the current-position is an unescaped `<` character
2. The current-position is equal to 0
3. All the following conditions are satisfied:
   1. The character at the current-position is not `<`, *and*
   2. The current-position is greater than 0, *and*
   3. The character at (current-position - 1) is a word-separator character

then the current-position is said to be a *potential-auto-link-start-position*.

If the current-position is a *potential-auto-link-start-position*, then the following is done:

1. If the remaining-character-sequence matches one of the following regular expression patterns (matching shall be case insensitive):
   1. URL within angle brackets: `/^<([a-z0-9\+\.\-]+:\/\/[^<> \`]+)>/`

      Example: `<http://example.net>`

   2. Mailto URL within angle brackets: `/^<(mailto:[^<> \`]+)>/`

      Example: `<mailto:someone@example.net?subject=Hi+there>`

   then the following is done:

   1. The matching substring for the whole of the matching pattern is identified as an **auto-link tag**

2. The matching substring for the first parenthesized subexpression in the matching pattern is called the *unprocessed auto-link url*. Any whitespace characters in the *unprocessed auto-link url* are removed, and the resultant string is called the *auto-link url*.

3. The output shall have a link with the link url set as *auto-link url* and the link text content also set as *auto-link url*. For HTML output, the link url should be URL-escaped and the link text should be html-text-escaped.

4. The consumed-character-count is set to the length of the *auto-link tag*

2. If the remaining-character-sequence matches the regular expression pattern `/^<([^\(\)\<\>\[\]\:\'\@\\\,\"\s\`]+@[^\(\)\<\>\[\]\:\'\@\\\,\"\s\`\.]+\.[^\(\)\<\>\[\]\:\'\@\\\,\"\s\`]+)>/` (Example: `<someone@example.net>`), then the following is done:

    1. The matching substring for the whole of the matching pattern is identified as an **auto-link tag**

    2. The matching substring for the first parenthesized subexpression in the matching pattern is called the *auto-link email*. Let the string formed by concatenating the string `mailto:` with the *auto-link email* be called as *auto-link email url*

    3. The output shall have a link with the link url set as *auto-link email url* and the link text content set as *auto-link email*. For HTML output, the link url should be URL-escaped and the link text should be html-text-escaped.

    4. The consumed-character-count is set to the length of the *auto-link tag*

3. If the remaining-character-sequence matches one of the following regular expression patterns (matching shall be greedy and case insensitive):

    1. URL without angle brackets: `/^([a-z0-9\+\.\-]+:\/\/)[^<>\`\s]+/`

       Example: `http://example.net`

    2. Mailto URL without angle brackets: `/^(mailto:)[^<>\`\s]+/`

       Example: `mailto:someone@example.net`

    then the following is done:

    1. The matching substring for the whole of the matching pattern is called the *unprocessed auto-link tag*

    2. The matching substring for the first and only parenthesized subexpression in the pattern is called the *auto-link scheme string*. The number of characters in the *auto-link scheme string* is called the *auto-link scheme string length*.

    3. The *unprocessed auto-link tag* shall be processed to give the *auto-link tag candidate*. The processing to be done is to remove any trailing speculative-url-end characters, such that the last character of the *auto-link tag candidate* is not a speculative-url-end character.

    4. If the length of the *auto-link tag candidate* is greater than the *auto-link scheme string length*, then the *auto-link tag candidate* is identified as an **auto-link tag**, and the following is done:

        1. The output shall have a link with the link url set as *auto-link tag candidate* and the link text content also set as *auto-link tag candidate*. For HTML output, the link url should be URL-escaped and the link text should be html-text-escaped.

        2. The consumed-character-count is set to the length of the *auto-link tag candidate*

    5. If the length of the *auto-link tag candidate* is lesser than or equal to the *auto-link scheme string length*, then the *auto-link scheme string* is identified as a *text fragment*, and the consumed-character-count is set to *auto-link scheme string length*

Procedure for identifying HTML tags

This procedure assumes that the character at the current-position is an unescaped `<` character.

We define a **phrasing-html-element** to be one of the following HTML elements: `a`, `abbr`, `area`, `audio`, `b`, `bdi`, `bdo`, `br`, `button`, `canvas`, `cite`, `code`, `data`, `datalist`, `del`, `dfn`, `em`, `embed`, `i`, `iframe`, `img`, `input`, `ins`, `kbd`, `keygen`, `label`, `map`, `mark`, `meter`, `noscript`, `object`, `output`, `progress`, `q`, `ruby`, `s`, `samp`, `select`, `small`, `span`, `strong`, `sub`, `sup`, `textarea`, `time`, `u`, `var`, `video` or `wbr`. These are elements in the HTML namespace that belong to the phrasing content category in HTML5.

We define a **verbatim-html-starter-tag-name** to be one of the following HTML tag names: `address`, `article`, `aside`, `blockquote`, `details`, `dialog`, `div`, `dl`, `fieldset`, `figure`, `footer`, `form`, `header`,

`main`, `nav`, `ol`, `section`, `table` or `ul`.

We define a **verbatim-html-container-tag-name** to be one of the following HTML tag names: `pre`, `script` or `style`.

Let *html-tag-detection-sequence* be the remaining-character-sequence.

To identify *span tags* related to inline HTML we need to employ the use of a HTML parser. We supply characters in the *html-tag-detection-sequence* to the HTML parser, one character at a time, till one of the following happens:

1. The HTML parser detects a complete HTML tag (start, end or self-closing tag) with tag name as either a verbatim-html-starter-tag-name or as a verbatim-html-container-tag-name.

   If this happens first, the following is done:

   1. The whole of the *html-tag-detection-sequence* is identified as *verbatim-html*
   2. For HTML output, this *verbatim-html* shall be included in the output verbatim (without subjecting it to the processing for text fragments)
   3. Set consumed-character-count to the length of the *html-tag-detection-sequence*

2. The HTML parser detects a complete self-closing HTML tag (this also includes tags empty by definition, like `<br>` and `<img src=&quot;picture.jpg&quot;>`, for example), and the name of the tag is neither a verbatim-html-starter-tag-name, nor a verbatim-html-container-tag-name.

   If this happens first, the following is done:

   1. The text that represents the self-closing HTML tag is identified as a **self-closing HTML tag**
   2. If the tag just identified is not a phrasing-html-element tag, then all nodes whose *node type* is not equal to *raw html node* are removed from the stack of potential opening span tags and interpreted as *text fragments*
   3. For HTML output, the text that represents the self-closing HTML tag shall be included in the output verbatim
   4. Set consumed-character-count to the length of the *self-closing HTML tag*

3. The HTML parser detects a complete opening HTML tag, and the name of the tag is neither a verbatim-html-starter-tag-name, nor a verbatim-html-container-tag-name.

   If this happens first, the following is done:

   1. The text that represents the opening HTML tag is identified as an **opening HTML tag**
   2. If the tag just identified is not a phrasing-html-element tag, then all nodes whose *node type* is not equal to *raw html node* are removed from the stack of potential opening span tags and interpreted as *text fragments*
   3. A new node is pushed onto the stack of potential opening span tags with the following properties:
      1. The *tag string* of the node is set to the text that represents the opening HTML tag
      2. The *node type* of the node is set as *raw html node*
      3. The *html tag name* of the node is set to the HTML tag name of the opening HTML tag that was just identified
   4. For HTML output, the text that represents the opening HTML tag shall be included in the output verbatim.
   5. Set consumed-character-count to the length of the *opening HTML tag*

4. The HTML parser detects a complete closing HTML tag, and the name of the tag is neither a verbatim-html-starter-tag-name, nor a verbatim-html-container-tag-name.

   If this happens first, the following is done:

   1. The text that represents the closing HTML tag is identified as a **closing HTML tag**
   2. If the tag just identified is not a phrasing-html-element tag, then all nodes whose *node type* is not equal to *raw html node* are removed from the stack of potential opening span tags and interpreted as *text fragments*
   3. Let *currently open html node* be the topmost node of type *raw html node*
   4. If the *currently open html node* is not *null*, and if the *html tag name* of the *currently open html node* is the same as the HTML tag name of the closing HTML tag that was just identified, the following is done:
      1. If the *currently open html node* is not already the top node, all nodes above it are popped off and interpreted as text fragments

2. The closing HTML tag that was just identified is considered as the matching tag for the HTML tag represented by the *currently open html node*

3. The *currently open html node* is popped off

5. If the *currently open html node* is *null*, or if *html tag name* of the *currently open html node* is not the same as the HTML tag name of the closing HTML tag that was just identified, then all nodes in the stack of potential opening span tags whose *node type* is not equal to *raw html node* shall be removed from the stack and interpreted as *text fragments*

6. For HTML output, the text that represents the closing HTML tag shall be included in the output verbatim.

7. Set consumed-character-count to the length of the *closing HTML tag*

5. The HTML parser detects a complete HTML comment.

   If this happens first, the text that represents the HTML comment is identified as a **comment HTML tag**.

   For HTML output, the text that represents the comment HTML tag shall be included in the output verbatim.

   consumed-character-count is set to the length of the *comment HTML tag*.

6. The HTML parser detects HTML text, or the HTML parser detects an error, or the *html-tag-detection-sequence* has no more characters to supply to the HTML parser.

   If this happens first, the `<` at the current-position is identified as a *text fragment*, and consumed-character-count is set to 1.

Additional processing

Some additional processing is required for certain parts before they can be written to the output.

De-escaping

Escaping backslashes in the input should not be part of the output.

We define a **punctuation** character to be a unicode code point whose 'General_Category' unicode property has one of the following values: Pc, Pd, Ps, Pe, Pi, Pf, Po.

We define a **symbol** character to be a unicode code point whose 'General_Category' unicode property has one of the following values: Sc, Sk, Sm, So.

To de-escape a string, every `\` (backslash) character in the string that is used for escaping a punctuation character or a symbol character, shall be removed.

For example, for the string `With \(esca\ped\) \\brackets`, the de-escaped string will be `With (esca\ped) \brackets`.

Processing text fragments

The *text fragments* identified in the procedure for identifying span tags are subject to the following processing:

1. The *text fragments* that occur adjacent to one another in the input character sequence are collated to form a single *collated text fragment*

2. Every *collated text fragment* is processed to produce a *processed text fragment*. The following processing is done:

   1. **Removing escaping backslashes:** The *collated text fragment* shall be de-escaped.

   2. **Introducing hard-breaks:** Every sequence of two space characters followed by a line break character shall be replaced by a hard line break as appropriate for the output format. For HTML output format, a `<br />` element is used to indicate a hard line break.

      For example, for the following *collated text fragment*:

      ```
      There are two spaces at the end of this line
      So we introduce a hard break there
      ```

      the corresponding *processed text fragment* for HTML output will be:

      ```
      There are two spaces at the end of this line<br />
      So we introduce a hard break there
      ```

   3. **HTML-related processing:** For HTML output, the *collated text fragment* shall be html-text-escaped.

For a non-HTML output format, any character references in the *collated text fragment* must be converted to a form appropriate to the output format. For many output formats, it might be appropriate to convert them to Unicode code points (for example, `&copy;` can be converted to U+00A9).

3. The *processed text fragment* is output

Processing for HTML output

For HTML output, the text that shall be output as part of text content of a HTML element, or the text that shall be output as part of a HTML tag's attribute value, needs to be escaped as described in this section.

HTML code escaping

The content of code blocks and code spans should be processed as follows before being output as HTML:

1. Replace the `<` character with `<`
2. Replace the `>` character with `>`
3. Replace the `&` character with `&`
4. Replace the `"` character with `&quot;`
5. Replace the `'` character with `&#x27;`

HTML text escaping

Text that forms the content of vfmd constructs other than code blocks and code spans should be processed as follows before being output as HTML:

1. Replace the `<` character with `<`
2. Replace the `>` character with `>`
3. Replace any `&` character with `&`, unless the `&` character forms the start of a character reference
4. Replace the `"` character with `&quot;`
5. Replace the `'` character with `&#x27;`

Attribute value escaping

Any text to be output as the value of a HTML attribute (other than the attributes mentioned in URL escaping) should be be processed as follows:

1. Replace the `<` character with `<`
2. Replace the `>` character with `>`
3. Replace any `&` character with `&`, unless the `&` character forms the start of a character reference
4. Replace the `"` character with `&quot;`
5. Replace the `'` character with `&#x27;`

When writing the HTML output for vfmd constructs, the `"` character should be used as the enclosing quote character for attribute values.

URL escaping

A URL that is output in HTML as either of the following:

1. As the value of a `href` attribute of an `a` tag

   (or)

2. As the value of a `src` attribute of an `img` tag

should be processed as follows before being output:

1. Percent-encode all bytes except the bytes that form the following ASCII characters:
   - Alphanumeric characters: `[A-Za-z0-9]`
   - Special characters: `$-_.+!*'(),`
   - Reserved characters: `;/?:@=&`

2. Replace any `&` character with `&`, unless the `&` character forms the start of a character reference

3. Replace the `’` character with `&#x27;`

Extending the syntax

An implementation can extend the core syntax to support additional syntax elements. The additional syntax elements can involve block-level extensions, or span-level extensions, or both.

For example, to support GitHub-style fenced code blocks, an implementation would need to add a block-level extension; to support GitHub-style strikethroughs, an implementation would need to add a span-level extension; to support MultiMarkdown-style footnotes, an implementation would need to add both a block-level extension (for handling footnote definitions) and a span-level extension (for handling footnote references).
