CSS Syntax Module Level 3

Editor's Draft, 9 October 2023

▼ **More details about this document**

This version:
https://drafts.csswg.org/css-syntax/

Latest published version:
https://www.w3.org/TR/css-syntax-3/

Previous Versions:
https://www.w3.org/TR/2019/CR-css-syntax-3-20190716/
https://www.w3.org/TR/2014/CR-css-syntax-3-20140220/
https://www.w3.org/TR/2013/WD-css-syntax-3-20131105/
https://www.w3.org/TR/2013/WD-css-syntax-3-20130919/

Feedback:
CSSWG Issues Repository

Editors:
Tab Atkins Jr. (Google)
Simon Sapin (Mozilla)

Suggest an Edit for this Spec:
GitHub Editor

Abstract

This module describes, in general terms, the basic structure and syntax of CSS stylesheets. It defines, in detail, the syntax and parsing of CSS - how to turn a stream of bytes into a meaningful stylesheet.

CSS is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, etc.

Status of this document

This is a public copy of the editors' draft. It is provided for discussion only and may change at any moment. Its publication here does not imply endorsement of its contents by W3C. Don't cite this document other than as work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-syntax" in the title, like this: "[css-syntax] *...summary of comment...*". All issues and comments are archived. Alternately, feedback can be sent to the (archived) public mailing list www-style@w3.org.

This document is governed by the 03 November 2023 W3C Process Document.

Table of Contents

§ 1. Introduction

This section is not normative.

This module defines the abstract syntax and parsing of CSS stylesheets and other things which use CSS syntax (such as the HTML `style` attribute).

It defines algorithms for converting a stream of Unicode code points (in other words, text) into a stream of CSS tokens, and then further into CSS objects such as stylesheets, rules, and declarations.

§ 1.1. Module interactions

This module defines the syntax and parsing of CSS stylesheets. It supersedes the lexical scanner and grammar defined in CSS 2.1.

§ 2. Description of CSS's Syntax

This section is not normative.

A CSS document is a series of style rules—which are qualified rules that apply styles to elements in a document—and at-rules—which define special processing rules or values for the CSS document.

A qualified rule starts with a prelude then has a {}-wrapped block containing a sequence of declarations. The meaning of the prelude varies based on the context that the rule appears in—for style rules, it's a selector which specifies what elements the declarations will apply to. Each declaration has a name, followed by a colon and the declaration value. Declarations are separated by semicolons.

> EXAMPLE 1 ¶
>
> A typical rule might look something like this:
>
> ```
> p > a {
>   color: blue;
>   text-decoration: underline;
> }
> ```
>
> In the above rule, "`p > a`" is the selector, which, if the source document is HTML, selects any `<a>` elements that are children of a `<p>` element.
>
> "`color: blue`" is a declaration specifying that, for the elements that match the selector, their 'color' property should have the value 'blue'. Similarly, their 'text-decoration' property should have the value 'underline'.

At-rules are all different, but they have a basic structure in common. They start with an "@" code point followed by their name as a CSS keyword. Some at-rules are simple statements, with their name followed by more CSS values to specify their behavior, and finally ended by a semicolon. Others are blocks; they can have CSS values following their name, but they end with a {}-wrapped block, similar to a qualified rule. Even the contents of these blocks are specific to the given at-rule: sometimes they contain a sequence of declarations, like a qualified rule; other times, they may contain additional blocks, or at-rules, or other structures altogether.

EXAMPLE 2

Here are several examples of at-rules that illustrate the varied syntax they may contain.

```
@import "my-styles.css";
```

The '@import' at-rule is a simple statement. After its name, it takes a single string or 'url()' function to indicate the stylesheet that it should import.

```
@page :left {
  margin-left: 4cm;
  margin-right: 3cm;
}
```

The '@page' at-rule consists of an optional page selector (the ':left' pseudoclass), followed by a block of properties that apply to the page when printed. In this way, it's very similar to a normal style rule, except that its properties don't apply to any "element", but rather the page itself.

```
@media print {
  body { font-size: 10pt }
}
```

The '@media' at-rule begins with a media type and a list of optional media queries. Its block contains entire rules, which are only applied when the '@media's conditions are fulfilled.

Property names and at-rule names are always ident sequences, which have to start with an ident-start code point, two hyphens, or a hyphen followed by an ident-start code point, and then can contain zero or more ident code points. You can include any code point at all, even ones that CSS uses in its syntax, by escaping it.

The syntax of selectors is defined in the Selectors spec. Similarly, the syntax of the wide variety of CSS values is defined in the Values & Units spec. The special syntaxes of individual at-rules can be found in the specs that define them.

2.1. Escaping

This section is not normative.

Any Unicode code point can be included in an ident sequence or quoted string by ***escaping*** it. CSS escape sequences start with a backslash (\), and continue with:

- Any Unicode code point that is not a hex digits or a newline. The escape sequence is replaced by that code point.
- Or one to six hex digits, followed by an optional whitespace. The escape sequence is replaced by the Unicode code point whose value is given by the hexadecimal digits. This optional whitespace allow hexadecimal escape sequences to be followed by "real" hex digits.

EXAMPLE 3
An ident sequence with the value "&B" could be written as '\26 B' or '\000026B'.

A "real" space after the escape sequence must be doubled.

2.2. Error Handling

This section is not normative.

When errors occur in CSS, the parser attempts to recover gracefully, throwing away only the minimum amount of content before returning to parsing as normal. This is because errors aren't always mistakes—new syntax looks like an error to an old parser, and it's useful to be able to add new syntax to the language without worrying about stylesheets that include it being completely broken in older UAs.

The precise error-recovery behavior is detailed in the parser itself, but it's simple enough that a short description is fairly accurate.

- At the "top level" of a stylesheet, an <at-keyword-token> starts an at-rule. Anything else starts a qualified rule, and is included in the rule's prelude. This may produce an invalid selector, but that's not the concern of the CSS parser—at worst, it means the selector will match nothing.
- Once an at-rule starts, nothing is invalid from the parser's standpoint; it's all part of the at-rule's prelude. Encountering a <semicolon-token> ends the at-rule immediately, while encountering an opening curly-brace <{-token> starts the at-rule's body. The at-rule seeks forward, matching blocks (content surrounded by (), {}, or []) until it finds a closing curly-brace <}-token> that isn't matched by anything else or inside of another block. The contents of the at-rule are then interpreted according to the at-rule's own grammar.
- Qualified rules work similarly, except that semicolons don't end them; instead, they are just taken in as part of the rule's prelude. When the first {} block is found, the contents are always interpreted as a list of declarations.
- When interpreting a list of declarations, unknown syntax at any point causes the parser to throw away whatever declaration it's currently building, and seek forward until it finds a semicolon (or the end of the block). It then starts fresh, trying to parse a declaration again.
- If the stylesheet ends while any rule, declaration, function, string, etc. are still open, everything is automatically closed. This doesn't make them invalid, though they may be incomplete and thus thrown away when they are verified against their grammar.

After each construct (declaration, style rule, at-rule) is parsed, the user agent checks it against its expected grammar. If it does not match the grammar, it's ***invalid***, and gets ***ignored*** by the UA, which treats it as if it wasn't there at all.

§ 3. Tokenizing and Parsing CSS

User agents must use the parsing rules described in this specification to generate the [CSSOM] trees from text/css resources. Together, these rules define what is referred to as the CSS parser.

This specification defines the parsing rules for CSS documents, whether they are syntactically correct or not. Certain points in the parsing algorithm are said to be ***parse errors***. The error handling for parse errors is well-defined: user agents must either act as described below when encountering such problems, or must abort processing at the first error that they encounter for which they do not wish to apply the rules described below.

Conformance checkers must report at least one parse error condition to the user if one or more parse error conditions exist in the document and must not report parse error conditions if none exist in the document. Conformance checkers may report more than one parse error condition if more than one parse error condition exists in the document. Conformance checkers are not required to recover from parse errors, but if they do, they must recover in the same way as user agents.

§ 3.1. Overview of the Parsing Model

The input to the CSS parsing process consists of a stream of Unicode code points, which is passed through a tokenization stage followed by a tree construction stage. The output is a CSSStyleSheet object.

> NOTE: Implementations that do not support scripting do not have to actually create a CSSOM CSSStyleSheet object, but the CSSOM tree in such cases is still used as the model for the rest of the specification.

§ 3.2. The input byte stream

When parsing a stylesheet, the stream of Unicode code points that comprises the input to the tokenization stage might be initially seen by the user agent as a stream of bytes (typically coming over the network or from the local file system). If so, the user agent must decode these bytes into code points according to a particular character encoding.

To ***decode*** a *stylesheet*'s stream of bytes into a stream of code points:

1. Determine the fallback encoding of *stylesheet*, and let *fallback* be the result.
2. Decode *stylesheet*'s stream of bytes with fallback encoding *fallback*, and return the result.

> NOTE: The decode algorithm gives precedence to a byte order mark (BOM), and only uses the fallback when none is found.

To ***determine the fallback encoding*** of a *stylesheet*:

1. If HTTP or equivalent protocol provides an *encoding label* (e.g. via the charset parameter of the Content-Type header) for the *stylesheet*, get an encoding from *encoding label*. If that does not return failure, return it.
2. Otherwise, check *stylesheet*'s byte stream. If the first 1024 bytes of the stream begin with the hex sequence

   ```
   40 63 68 61 72 73 65 74 20 22 XX* 22 3B
   ```

   where each `XX` byte is a value between 0_{16} and 21_{16} inclusive or a value between 23_{16} and $7F_{16}$ inclusive, then get an encoding from a string formed out of the sequence of `XX` bytes, interpreted as `ASCII`.

   > ▶ **What does that byte sequence mean?**

   If the return value was `utf-16be` or `utf-16le`, return `utf-8`; if it was anything else except failure, return it.

   > ▶ **Why use utf-8 when the declaration says utf-16?**

   > NOTE: Note that the syntax of an encoding declaration *looks like* the syntax of an at-rule named '@charset', but no such rule actually exists, and the rules for how you can write it are much more restrictive than they would normally be for recognizing such a rule. A number of things you can do in CSS that would produce a valid '@charset' rule (if one existed), such as using multiple spaces, comments, or single quotes, will cause the encoding declaration to not be recognized. This behavior keeps the encoding declaration as simple as possible, and thus maximizes the likelihood of it being implemented correctly.

3. Otherwise, if an environment encoding is provided by the referring document, return it.
4. Otherwise, return `utf-8`.

> Though UTF-8 is the default encoding for the web, and many newer web-based file formats assume or require UTF-8 encoding, CSS was created before it was clear which encoding would win, and thus can't automatically assume the stylesheet is UTF-8.
>
> Stylesheet authors *should* author their stylesheets in UTF-8, and ensure that either an HTTP header (or equivalent method) declares the encoding of the stylesheet to be UTF-8, or that the referring document declares its encoding to be UTF-8. (In HTML, this is done by adding a `<meta charset=utf-8>` element to the head of the document.)
>
> If neither of these options are available, authors should begin the stylesheet with a UTF-8 BOM or the exact characters
>
> `@charset "utf-8";`

Document languages that refer to CSS stylesheets that are decoded from bytes may define an ***environment encoding*** for each such stylesheet, which is used as a fallback when other encoding hints are not available or can not be used.

The concept of environment encoding only exists for compatibility with legacy content. New formats and new linking mechanisms **should not** provide an environment encoding, so the stylesheet defaults to UTF-8 instead in the absence of more explicit information.

> NOTE: [HTML] defines the environment encoding for `<link rel=stylesheet>`.

> NOTE: [CSSOM] defines the environment encoding for `<xml-stylesheet?>`.

> NOTE: [CSS-CASCADE-3] defines 'the environment encoding for `@import`'.

§ 3.3. Preprocessing the input stream

The ***input stream*** consists of the filtered code points pushed into it as the input byte stream is decoded.

To ***filter code points*** from a stream of (unfiltered) code points *input*:

- Replace any U+000D CARRIAGE RETURN (CR) code points, U+000C FORM FEED (FF) code points, or pairs of U+000D CARRIAGE RETURN (CR) followed by U+000A LINE FEED (LF) in *input* by a single U+000A LINE FEED (LF) code point.
- Replace any U+0000 NULL or surrogate code points in *input* with U+FFFD REPLACEMENT CHARACTER ().

> NOTE: The only way to produce a surrogate code point in CSS content is by directly assigning a `DOMString` with one in it via an OM operation.

§ 4. Tokenization

To ***tokenize*** a stream of code points into a stream of CSS tokens *input*, repeatedly consume a token from *input* until an <EOF-token> is reached, pushing each of the returned tokens into a stream.

> NOTE: Each call to the consume a token algorithm returns a single token, so it can also be used "on-demand" to tokenize a stream of code points *during* parsing, if so desired.

The output of tokenization step is a stream of zero or more of the following tokens: ***'<ident-token>'***, ***'<function-token>'***, ***'<at-keyword-token>'***, ***'<hash-token>'***, ***'<string-token>'***, ***'<bad-string-token>'***, ***'<url-token>'***, ***'<bad-url-token>'***, ***'<delim-token>'***, ***'<number-token>'***, ***'<percentage-token>'***, ***'<dimension-token>'***, ***'<unicode-range-token>'***, ***'<whitespace-token>'***, ***'<CDO-token>'***, ***'<CDC-token>'***, ***'<colon-token>'***, ***'<semicolon-token>'***, ***'<comma-token>'***, ***'<[-token>'***, ***'<]-token>'***, ***'<(-token>'***, ***'<)-token>'***, ***'<{-token>'***, and ***'<}-token>'***.

- <ident-token>, <function-token>, <at-keyword-token>, <hash-token>, <string-token>, and <url-token> have a value composed of zero or more code points. Additionally, hash tokens have a type flag set to either "id" or "unrestricted". The type flag defaults to "unrestricted" if not otherwise set.
- <delim-token> has a value composed of a single code point.
- <number-token>, <percentage-token>, and <dimension-token> have a numeric value, and an optional sign character set to either "+" or "-" (or nothing).

  <number-token> and <dimension-token> additionally have a type flag set to either "integer" or "number". The type flag defaults to "integer" if not otherwise set. <dimension-token> additionally have a unit composed of one or more code

points.

- <unicode-range-token> has a starting and ending code point. It represents an inclusive range of codepoints (including both the start and end). If the ending code point is before the starting code point, it represents an empty range.

> NOTE: The type flag of hash tokens is used in the Selectors syntax [SELECT]. Only hash tokens with the "id" type are valid ID selectors.

§ 4.1. Token Railroad Diagrams

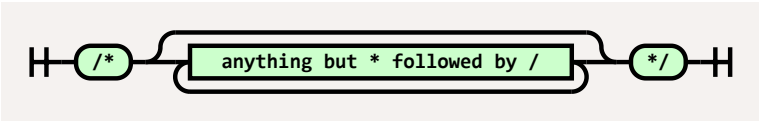
This section is non-normative.

This section presents an informative view of the tokenizer, in the form of railroad diagrams. Railroad diagrams are more compact than an explicit parser, but often easier to read than an regular expression.

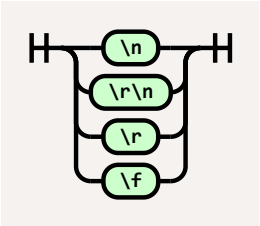
These diagrams are *informative* and *incomplete*; they describe the grammar of "correct" tokens, but do not describe error-handling at all. They are provided solely to make it easier to get an intuitive grasp of the syntax of each token.

Diagrams with names such as <*foo-token*> represent tokens. The rest are productions referred to by other diagrams.

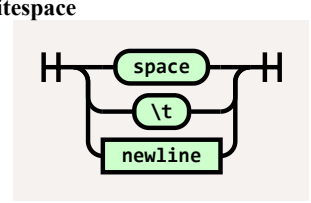
¶ **comment**



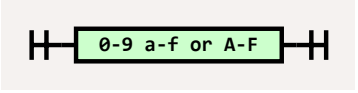
¶ **newline**



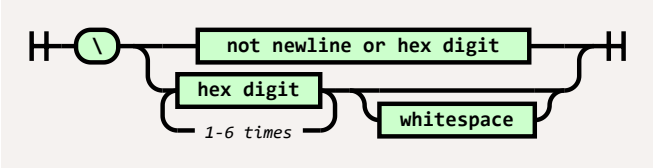
¶ **whitespace**



¶ **hex digit**



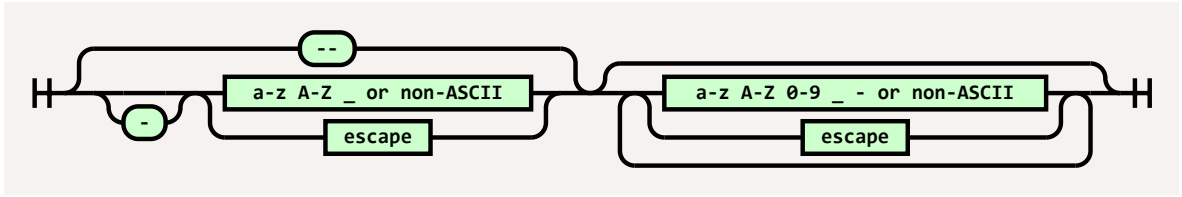
¶ **escape**



¶ **<whitespace-token>**

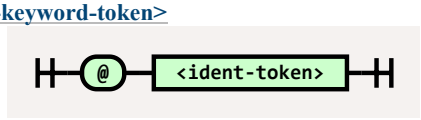
¶ **ws***

¶ **<ident-token>**

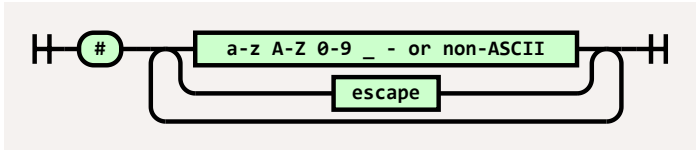


¶ **<function-token>**

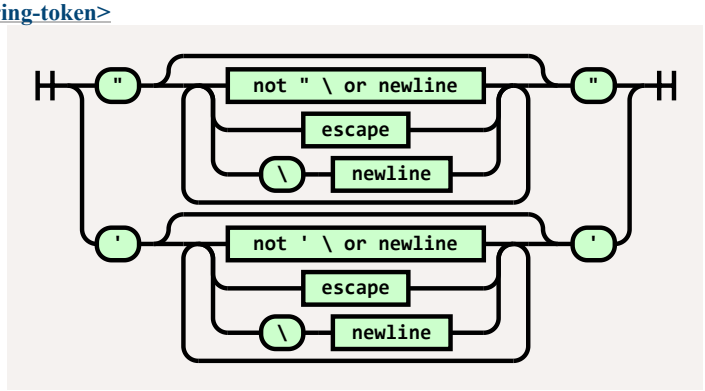
¶ **<at-keyword-token>**



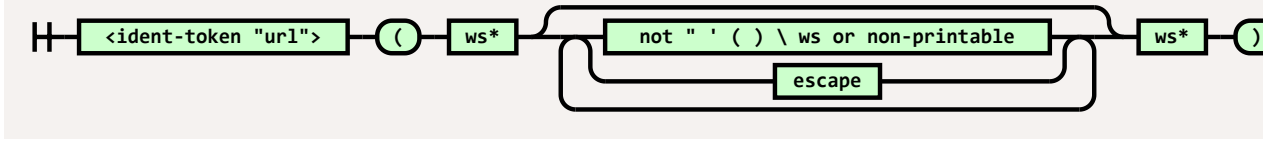
¶ **<hash-token>**



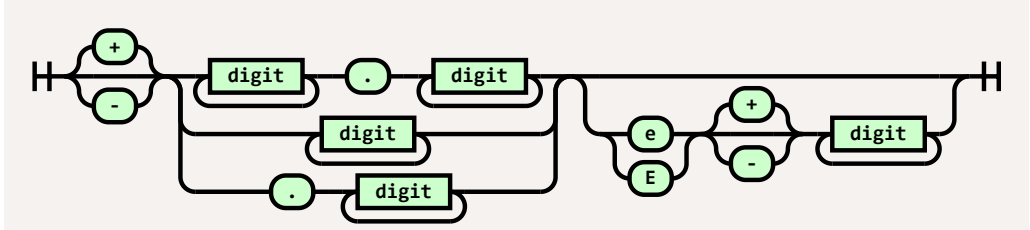
¶ **<string-token>**



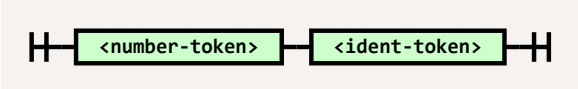
¶ **<url-token>**



¶ **<number-token>**



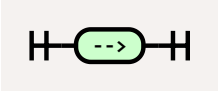
¶ **<dimension-token>**



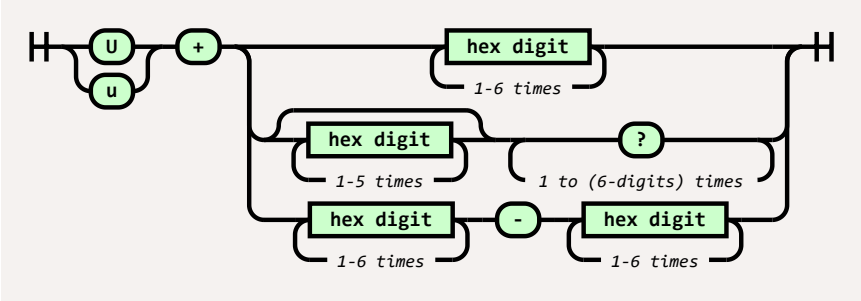
¶ **<percentage-token>**

¶ **<CDO-token>**

¶ **<CDC-token>**



¶ **<unicode-range-token>**



§ 4.2. Definitions

This section defines several terms used during the tokenization phase.

next input code point
: The first code point in the input stream that has not yet been consumed.

current input code point
: The last code point to have been consumed.

reconsume the current input code point
: Push the current input code point back onto the front of the input stream, so that the next time you are instructed to consume the next input code point, it will instead reconsume the current input code point.

EOF code point
: A conceptual code point representing the end of the input stream. Whenever the input stream is empty, the next input code point is always an EOF code point.

digit
: A code point between U+0030 DIGIT ZERO (0) and U+0039 DIGIT NINE (9) inclusive.

hex digit
: A digit, or a code point between U+0041 LATIN CAPITAL LETTER A (A) and U+0046 LATIN CAPITAL LETTER F (F) inclusive, or a code point between U+0061 LATIN SMALL LETTER A (a) and U+0066 LATIN SMALL LETTER F (f) inclusive.

uppercase letter
: A code point between U+0041 LATIN CAPITAL LETTER A (A) and U+005A LATIN CAPITAL LETTER Z (Z) inclusive.

lowercase letter
: A code point between U+0061 LATIN SMALL LETTER A (a) and U+007A LATIN SMALL LETTER Z (z) inclusive.

letter
An uppercase letter or a lowercase letter.

non-ASCII ident code point
A code point whose value is any of:

- U+00B7
- between U+00C0 and U+00D6
- between U+00D8 and U+00F6
- between U+00F8 and U+037D
- between U+037F and U+1FFF
- U+200C
- U+200D
- U+203F
- U+2040
- between U+2070 and U+218F
- between U+2C00 and U+2FEF
- between U+3001 and U+D7FF
- between U+F900 and U+FDCF
- between U+FDF0 and U+FFFD
- greater than or equal to U+10000

All of these ranges are inclusive.

▶ **Why these character, specifically?**

ident-start code point
A letter, a non-ASCII ident code point, or U+005F LOW LINE (_).

ident code point
An ident-start code point, a digit, or U+002D HYPHEN-MINUS (-).

non-printable code point
A code point between U+0000 NULL and U+0008 BACKSPACE inclusive, or U+000B LINE TABULATION, or a code point between U+000E SHIFT OUT and U+001F INFORMATION SEPARATOR ONE inclusive, or U+007F DELETE.

newline
U+000A LINE FEED. Note that U+000D CARRIAGE RETURN and U+000C FORM FEED are not included in this definition, as they are converted to U+000A LINE FEED during preprocessing.

whitespace
A newline, U+0009 CHARACTER TABULATION, or U+0020 SPACE.

maximum allowed code point
The greatest code point defined by Unicode: U+10FFFF.

ident sequence
A sequence of code points that has the same syntax as an <ident-token>.

NOTE: The part of an <at-keyword-token> after the "@", the part of a <hash-token> (with the "id" type flag) after the "#", the part of a <function-token> before the "(", and the unit of a <dimension-token> are all ident sequences.

§ 4.3. Tokenizer Algorithms

The algorithms defined in this section transform a stream of code points into a stream of tokens.

§ 4.3.1. Consume a token

This section describes how to ***consume a token*** from a stream of code points. It additionally takes an optional boolean *unicode ranges allowed*, defaulting to false. It will return a single token of any type.

Consume comments.

Consume the next input code point.

whitespace
: Consume as much whitespace as possible. Return a <whitespace-token>.

U+0022 QUOTATION MARK (")
: Consume a string token and return it.

U+0023 NUMBER SIGN (#)
: If the next input code point is an ident code point or the next two input code points are a valid escape, then:

1. Create a <hash-token>.
2. If the next 3 input code points would start an ident sequence, set the <hash-token>'s type flag to "id".
3. Consume an ident sequence, and set the <hash-token>'s value to the returned string.
4. Return the <hash-token>.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+0027 APOSTROPHE (')
: Consume a string token and return it.

U+0028 LEFT PARENTHESIS (()
: Return a <(-token>.

U+0029 RIGHT PARENTHESIS ())
: Return a <)-token>.

U+002B PLUS SIGN (+)
: If the input stream starts with a number, reconsume the current input code point, consume a numeric token, and return it.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+002C COMMA (,)
: Return a <comma-token>.

U+002D HYPHEN-MINUS (-)
: If the input stream starts with a number, reconsume the current input code point, consume a numeric token, and return it.

Otherwise, if the next 2 input code points are U+002D HYPHEN-MINUS U+003E GREATER-THAN SIGN (->), consume them and return a <CDC-token>.

Otherwise, if the input stream starts with an ident sequence, reconsume the current input code point, consume an ident-like token, and return it.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+002E FULL STOP (.)
If the input stream starts with a number, reconsume the current input code point, consume a numeric token, and return it.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+003A COLON (:)
Return a <colon-token>.

U+003B SEMICOLON (;)
Return a <semicolon-token>.

U+003C LESS-THAN SIGN (<)
If the next 3 input code points are U+0021 EXCLAMATION MARK U+002D HYPHEN-MINUS U+002D HYPHEN-MINUS (!--), consume them and return a <CDO-token>.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+0040 COMMERCIAL AT (@)
If the next 3 input code points would start an ident sequence, consume an ident sequence, create an <at-keyword-token> with its value set to the returned value, and return it.

Otherwise, return a <delim-token> with its value set to the current input code point.

U+005B LEFT SQUARE BRACKET ([)
Return a <[-token>.

U+005C REVERSE SOLIDUS (\)
If the input stream starts with a valid escape, reconsume the current input code point, consume an ident-like token, and return it.

Otherwise, this is a parse error. Return a <delim-token> with its value set to the current input code point.

U+005D RIGHT SQUARE BRACKET (])
Return a <]-token>.

U+007B LEFT CURLY BRACKET ({)
Return a <{-token>.

U+007D RIGHT CURLY BRACKET (})
Return a <}-token>.

digit
Reconsume the current input code point, consume a numeric token, and return it.

U+0055 LATIN CAPITAL LETTER U (U)
u+0075 LATIN LOWERCASE LETTER U (u)
If *unicode ranges allowed* is true and the input stream would start a unicode-range, reconsume the current input code point, consume a unicode-range token, and return it.

Otherwise, reconsume the current input code point, consume an ident-like token, and return it.

ident-start code point
Reconsume the current input code point, consume an ident-like token, and return it.

EOF
Return an <EOF-token>.

anything else
Return a <delim-token> with its value set to the current input code point.

§ 4.3.2. Consume comments

This section describes how to ***consume comments*** from a stream of code points. It returns nothing.

If the next two input code point are U+002F SOLIDUS (/) followed by a U+002A ASTERISK (*), consume them and all following code points up to and including the first U+002A ASTERISK (*) followed by a U+002F SOLIDUS (/), or up to an EOF code point. Return to the start of this step.

If the preceding paragraph ended by consuming an EOF code point, this is a parse error.

Return nothing.

§ 4.3.3. Consume a numeric token

This section describes how to ***consume a numeric token*** from a stream of code points. It returns either a <number-token>, <percentage-token>, or <dimension-token>.

Consume a number and let *number* be the result.

If the next 3 input code points would start an ident sequence, then:

1. Create a <dimension-token> with the same value, type flag, and sign character as *number*, and a unit set initially to the empty string.
2. Consume an ident sequence. Set the <dimension-token>'s unit to the returned value.
3. Return the <dimension-token>.

Otherwise, if the next input code point is U+0025 PERCENTAGE SIGN (%), consume it. Create a <percentage-token> with the same value and sign character as *number*, and return it.

Otherwise, create a <number-token> with the same value, type flag, and sign character as *number*, and return it.

§ 4.3.4. Consume an ident-like token

This section describes how to ***consume an ident-like token*** from a stream of code points. It returns an <ident-token>, <function-token>, <url-token>, or <bad-url-token>.

Consume an ident sequence, and let *string* be the result.

If *string*'s value is an ASCII case-insensitive match for "url", and the next input code point is U+0028 LEFT PARENTHESIS (), consume it. While the next two input code points are whitespace, consume the next input code point. If the next one or two input code points are U+0022 QUOTATION MARK ("), U+0027 APOSTROPHE ('), or whitespace followed by U+0022 QUOTATION MARK (") or U+0027 APOSTROPHE ('), then create a <function-token> with its value set to *string* and return it. Otherwise, consume a url token, and return it.

Otherwise, if the next input code point is U+0028 LEFT PARENTHESIS (), consume it. Create a <function-token> with its value set to *string* and return it.

Otherwise, create an <ident-token> with its value set to *string* and return it.

§ 4.3.5. Consume a string token

This section describes how to ***consume a string token*** from a stream of code points. It returns either a <string-token> or <bad-string-token>.

This algorithm may be called with an *ending code point*, which denotes the code point that ends the string. If an *ending code point* is not specified, the current input code point is used.

Initially create a <string-token> with its value set to the empty string.

Repeatedly consume the next input code point from the stream:

ending code point
Return the <string-token>.

EOF
This is a parse error. Return the <string-token>.

newline
This is a parse error. Reconsume the current input code point, create a <bad-string-token>, and return it.

U+005C REVERSE SOLIDUS (\)
If the next input code point is EOF, do nothing.

Otherwise, if the next input code point is a newline, consume it.

Otherwise, (the stream starts with a valid escape) consume an escaped code point and append the returned code point to the <string-token>'s value.

anything else
Append the current input code point to the <string-token>'s value.

§ 4.3.6. Consume a url token

This section describes how to ***consume a url token*** from a stream of code points. It returns either a <url-token> or a <bad-url-token>.

> NOTE: This algorithm assumes that the initial "url(" has already been consumed. This algorithm also assumes that it's being called to consume an "unquoted" value, like 'url(foo)'. A quoted value, like 'url("foo")', is parsed as a <function-token>. Consume an ident-like token automatically handles this distinction; this algorithm shouldn't be called directly otherwise.

1. Initially create a <url-token> with its value set to the empty string.
2. Consume as much whitespace as possible.
3. Repeatedly consume the next input code point from the stream:

   U+0029 RIGHT PARENTHESIS ())
   Return the <url-token>.

   EOF
   This is a parse error. Return the <url-token>.

   whitespace
   Consume as much whitespace as possible. If the next input code point is U+0029 RIGHT PARENTHESIS ()) or EOF, consume it and return the <url-token> (if EOF was encountered, this is a parse error); otherwise, consume the remnants of a bad url, create a <bad-url-token>, and return it.

   U+0022 QUOTATION MARK (")
   U+0027 APOSTROPHE (')
   U+0028 LEFT PARENTHESIS (()
   non-printable code point
   This is a parse error. Consume the remnants of a bad url, create a <bad-url-token>, and return it.

   U+005C REVERSE SOLIDUS (\)
   If the stream starts with a valid escape, consume an escaped code point and append the returned code point to the <url-token>'s value.

   Otherwise, this is a parse error. Consume the remnants of a bad url, create a <bad-url-token>, and return it.

   anything else
   Append the current input code point to the <url-token>'s value.

§ 4.3.7. Consume an escaped code point

This section describes how to ***consume an escaped code point***. It assumes that the U+005C REVERSE SOLIDUS (\) has already been consumed and that the next input code point has already been verified to be part of a valid escape. It will return a code point.

Consume the next input code point.

hex digit
: Consume as many hex digits as possible, but no more than 5. Note that this means 1-6 hex digits have been consumed in total. If the next input code point is whitespace, consume it as well. Interpret the hex digits as a hexadecimal number. If this number is zero, or is for a surrogate, or is greater than the maximum allowed code point, return U+FFFD REPLACEMENT CHARACTER (�). Otherwise, return the code point with that value.

EOF
: This is a parse error. Return U+FFFD REPLACEMENT CHARACTER (�).

anything else
: Return the current input code point.

§ 4.3.8. Check if two code points are a valid escape

This section describes how to ***check if two code points are a valid escape***. The algorithm described here can be called explicitly with two code points, or can be called with the input stream itself. In the latter case, the two code points in question are the current input code point and the next input code point, in that order.

> NOTE: This algorithm will not consume any additional code point.

If the first code point is not U+005C REVERSE SOLIDUS (\), return false.

Otherwise, if the second code point is a newline, return false.

Otherwise, return true.

§ 4.3.9. Check if three code points would start an ident sequence

This section describes how to ***check if three code points would start an ident sequence***. The algorithm described here can be called explicitly with three code points, or can be called with the input stream itself. In the latter case, the three code points in question are the current input code point and the next two input code points, in that order.

> NOTE: This algorithm will not consume any additional code points.

Look at the first code point:

U+002D HYPHEN-MINUS
: If the second code point is an ident-start code point or a U+002D HYPHEN-MINUS, or the second and third code points are a valid escape, return true. Otherwise, return false.

ident-start code point
: Return true.

U+005C REVERSE SOLIDUS (\)
: If the first and second code points are a valid escape, return true. Otherwise, return false.

anything else
: Return false.

§ 4.3.10. Check if three code points would start a number

This section describes how to ***check if three code points would start a number***. The algorithm described here can be called explicitly with three code points, or can be called with the input stream itself. In the latter case, the three code points in question are the current input code point and the next two input code points, in that order.

> NOTE: This algorithm will not consume any additional code points.

Look at the first code point:

U+002B PLUS SIGN (+)
U+002D HYPHEN-MINUS (-)
If the second code point is a digit, return true.

Otherwise, if the second code point is a U+002E FULL STOP (.) and the third code point is a digit, return true.

Otherwise, return false.

U+002E FULL STOP (.)
If the second code point is a digit, return true. Otherwise, return false.

digit
Return true.

anything else
Return false.

§ 4.3.11. Check if three code points would start a unicode-range

This section describes how to ***check if three code points would start a unicode-range***. The algorithm described here can be called explicitly with three code points, or can be called with the input stream itself. In the latter case, the three code points in question are the current input code point and the next two input code points, in that order.

> NOTE: This algorithm will not consume any additional code points.

If all of the following are true:

- The first code point is either U+0055 LATIN CAPITAL LETTER U (U) or U+0075 LATIN SMALL LETTER U (u)
- The second code point is U+002B PLUS SIGN (+).
- The third code point is either U+003F QUESTION MARK (?) or a hex digit

then return true.

Otherwise return false.

§ 4.3.12. Consume an ident sequence

This section describes how to ***consume an ident sequence*** from a stream of code points. It returns a string containing the largest name that can be formed from adjacent code points in the stream, starting from the first.

> NOTE: This algorithm does not do the verification of the first few code points that are necessary to ensure the returned code points would constitute an <ident-token>. If that is the intended use, ensure that the stream starts with an ident sequence before calling this algorithm.

Let *result* initially be an empty string.

Repeatedly consume the next input code point from the stream:

ident code point
: Append the code point to *result*.

the stream starts with a valid escape
: Consume an escaped code point. Append the returned code point to *result*.

anything else
: Reconsume the current input code point. Return *result*.

§ 4.3.13. Consume a number

This section describes how to ***consume a number*** from a stream of code points. It returns a numeric *value*, a string *type* which is either "integer" or "number", and an optional *sign character* which is either "+", "-", or missing.

> NOTE: This algorithm does not do the verification of the first few code points that are necessary to ensure a number can be obtained from the stream. Ensure that the stream starts with a number before calling this algorithm.

Execute the following steps in order:

1. Let *type* be the string "integer". Let *number part* and *exponent part* be the empty string.
2. If the next input code point is U+002B PLUS SIGN (+) or U+002D HYPHEN-MINUS (-), consume it. Append it to *number part* and set *sign character* to it.
3. While the next input code point is a digit, consume it and append it to *number part*.
4. If the next 2 input code points are U+002E FULL STOP (.) followed by a digit, then:
    1. Consume the next input code point and append it to *number part*.
    2. While the next input code point is a digit, consume it and append it to *number part*.
    3. Set *type* to "number".
5. If the next 2 or 3 input code points are U+0045 LATIN CAPITAL LETTER E (E) or U+0065 LATIN SMALL LETTER E (e), optionally followed by U+002D HYPHEN-MINUS (-) or U+002B PLUS SIGN (+), followed by a digit, then:
    1. Consume the next input code point.
    2. If the next input code point is "+" or "-", consume it and append it to *exponent part*.
    3. While the next input code point is a digit, consume it and append it to *exponent part*.
    4. Set *type* to "number".
6. Let *value* be the result of interpreting *number part* as a base-10 number.

    If *exponent part* is non-empty, interpret it as a base-10 integer, then raise 10 to the power of the result, multiply it by *value*, and set *value* to that result.

7. Return *value*, *type*, and *sign character*.

§ 4.3.14. Consume a unicode-range token

This section describes how to ***consume a unicode-range token*** from a stream of code points. It returns a <unicode-range-token>.

> NOTE: This algorithm does not do the verification of the first few code points that are necessary to ensure the returned code points would constitute an <unicode-range-token>. Ensure that the stream would start a unicode-range before calling this algorithm.

NOTE: This token is not produced by the tokenizer under normal circumstances. This algorithm is only called during consume the value of a unicode-range descriptor, which itself is only called as a special case for parsing the 'unicode-range' descriptor; this single invocation in the entire language is due to a bad syntax design in early CSS.

1. Consume the next two input code points and discard them.
2. Consume as many hex digits as possible, but no more than 6. If less than 6 hex digits were consumed, consume as many U+003F QUESTION MARK (?) code points as possible, but no more than enough to make the total of hex digits and U+003F QUESTION MARK (?) code points equal to 6.

   Let *first segment* be the consumed code points.
3. If *first segment* contains any question mark code points, then:
   1. Replace the question marks in *first segment* with U+0030 DIGIT ZERO (0) code points, and interpret the result as a hexadecimal number. Let this be *start of range*.
   2. Replace the question marks in *first segment* with U+0046 LATIN CAPITAL LETTER F (F) code points, and interpret the result as a hexadecimal number. Let this be *end of range*.
   3. Return a new <unicode-range-token> starting at *start of range* and ending at *end of range*.
4. Otherwise, interpret *first segment* as a hexadecimal number, and let the result be *start of range*.
5. If the next 2 input code points are U+002D HYPHEN-MINUS (-) followed by a hex digit, then:
   1. Consume the next input code point.
   2. Consume as many hex digits as possible, but no more than 6. Interpret the consumed code points as a hexadecimal number. Let this be *end of range*.
   3. Return a new <unicode-range-token> starting at *start of range* and ending at *end of range*.
6. Otherwise, return a new <unicode-range-token> both starting and ending at *start of range*.

4.3.15. Consume the remnants of a bad url

This section describes how to ***consume the remnants of a bad url*** from a stream of code points, "cleaning up" after the tokenizer realizes that it's in the middle of a <bad-url-token> rather than a <url-token>. It returns nothing; its sole use is to consume enough of the input stream to reach a recovery point where normal tokenizing can resume.

Repeatedly consume the next input code point from the stream:

U+0029 RIGHT PARENTHESIS ())
EOF
: Return.

the input stream starts with a valid escape
: Consume an escaped code point. This allows an escaped right parenthesis ("\)") to be encountered without ending the <bad-url-token>. This is otherwise identical to the "anything else" clause.

anything else
: Do nothing.

5. Parsing

The CSS parser converts a token stream (produced by the tokenization process, defined earlier in this spec) into one or more of several CSS constructs (depending on which parsing algorithm is invoked).

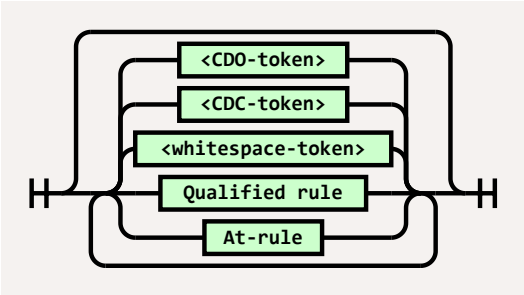
§ 5.1. Parser Railroad Diagrams

This section is non-normative.

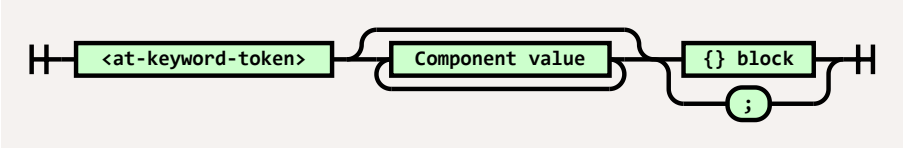
This section presents an informative view of the parser, in the form of railroad diagrams.

These diagrams are *informative* and *incomplete*; they describe the grammar of "correct" stylesheets, but do not describe error-handling at all. They are provided solely to make it easier to get an intuitive grasp of the syntax.

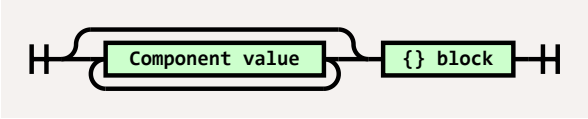
¶ Stylesheet



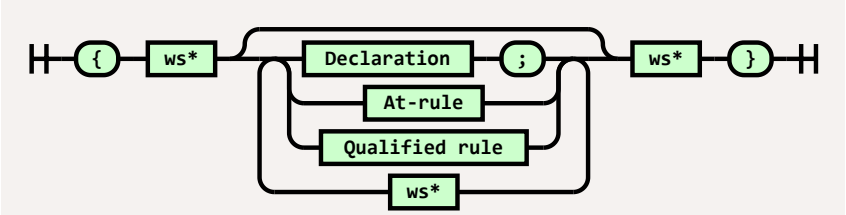
¶ At-rule



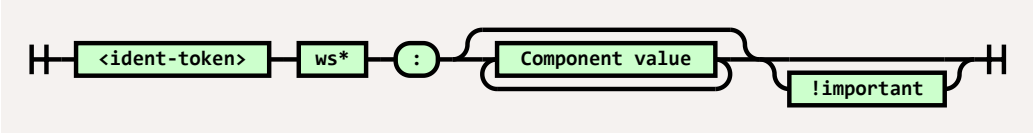
¶ Qualified rule



¶ {} block

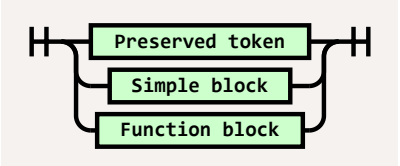


¶ Declaration

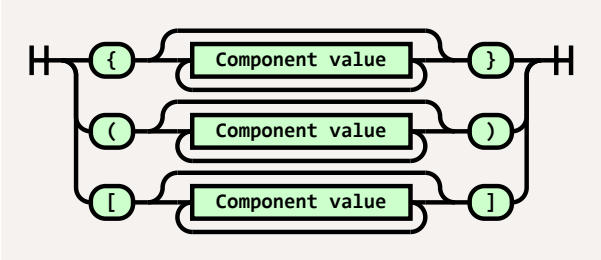


¶ !important

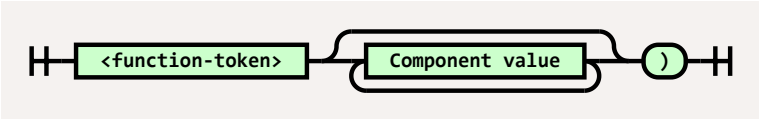
¶ Component value



¶ **Simple block**



¶ **Function block**



§ 5.2. CSS Parsing Results

The result of parsing can be any of the following (or lists of these):

stylesheet
A stylesheet has a list of rules.

rule
A rule is either an at-rule or a qualified rule.

at-rule
An at-rule has a name which is a string, a prelude consisting of a list of component values. Block at-rules (ending in a {}-block) will additionally have a list of declarations and a list of child rules. (Statement at-rules, ending in a semicolon, will not.)

qualified rule
A qualified rule has a prelude consisting of a list of component values, a list of declarations, and a list of child rules.

> NOTE: Most qualified rules will be style rules, where the prelude is a selector [selectors-4] and its declarations are properties.

declaration
A declaration has a name which is a string, a value consisting of a list of component values, and an *important* flag which is initially unset. It also has an optional *original text* which is a string (captured for only a few declarations).

Declarations are further categorized as ***property declarations*** or ***descriptor declarations***, with the former setting CSS properties and appearing most often in qualified rules and the latter setting CSS descriptors, which appear only in at-rules. (This categorization does not occur at the Syntax level; instead, it is a product of where the declaration appears, and is defined by the respective specifications defining the given rule.)

component value
A component value is one of the preserved tokens, a function, or a simple block.

preserved tokens
Any token produced by the tokenizer except for <function-token>s, <{-token>s, <(-token>s, and <[-token>s.

> NOTE: The non-preserved tokens listed above are always consumed into higher-level objects, either functions or simple blocks, and so never appear in any parser output themselves.

> NOTE: The tokens <}-token>s, <)-token>s, <]-token>, <bad-string-token>, and <bad-url-token> are always parse errors, but they are preserved in the token stream by this specification to allow other specs, such as Media Queries, to define more fine-grained error-handling than just dropping an entire declaration or block.

function
A function has a name and a value consisting of a list of component values.

simple block
{}-block
[]-block
()-block
A simple block has an associated token (either a <[-token>, <(-token>, or <{-token>) and a value consisting of a list of component values.

{}-block, []-block, and ()-block refer specifically to a simple block with that corresponding associated token.

§ 5.3. Token Streams

A ***token stream*** is a struct representing a stream of tokens and/or component values. It has the following items:

tokens
A list of tokens and/or component values.

> NOTE: This specification assumes, for simplicity, that the input stream has been fully tokenized before parsing begins. However, the parsing algorithms only use one token of "lookahead", so in practice tokenization and parsing can be done in lockstep.

index
An index into the tokens, representing the progress of parsing. It starts at 0 initially.

> NOTE: Aside from marking, the index never goes backwards. Thus the already-processed prefix of tokens can be eagerly discarded as it's processed.

marked indexes
A stack of index values, representing points that the parser might return to. It starts empty initially.

CSS has a small number of places that require referencing the precise text that was parsed for a declaration's value (not just what tokens were produced from that text). This is not explicitly described in the algorithmic structure here, but the token stream must, thus, have the ability to reproduce the original text of declarations on demand. See consume a declaration for details on when this is required.

Several operations can be performed on a token stream:

next token
The item of tokens at index.

If that index would be out-of-bounds past the end of the list, it's instead an <eof-token>.

empty
A token stream is empty if the next token is an <eof-token>.

consume a token
Let *token* be the next token. Increment index, then return *token*.

discard a token
If the token stream is not empty, increment index.

mark
Append index to marked indexes.

restore a mark
Pop from marked indexes, and set index to the popped value.

discard a mark
Pop from marked indexes, and do nothing with the popped value.

discard whitespace
: While the next token is a <whitespace-token>, discard a token.

process
: To process, given a following list of token types and associated actions, perform the action associated with the next token. Repeat until one of the actions returns something, then return that.

An ***'<eof-token>'*** is a conceptual token, not actually produced by the tokenizer, used to indicate that the token stream has been exhausted.

§ 5.4. Parser Entry Points

The algorithms defined in this section produce high-level CSS objects from lists of CSS tokens.

The algorithms here operate on a token stream as input, but for convenience can also be invoked with a number of other value types.

To ***normalize into a token stream*** a given *input*:

1. If *input* is already a token stream, return it.
2. If *input* is a list of CSS tokens and/or component values, create a new token stream with *input* as its tokens, and return it.
3. If *input* is a string, then filter code points from *input*, tokenize the result, then create a new token stream with those tokens as its tokens, and return it.
4. Assert: Only the preceding types should be passed as *input*.

NOTE: Other specs can define additional entry points for their own purposes.

The following notes should probably be translated into normative text in the relevant specs, hooking this spec's terms:

- "Parse a stylesheet" is intended to be the normal parser entry point, for parsing stylesheets.
- "Parse a stylesheet's contents" is intended for use by the `CSSStyleSheet replace()` method, and similar, which parse text into the contents of an existing stylesheet.
- "Parse a rule" is intended for use by the `CSSStyleSheet insertRule()` method, and similar, which parse text into a single rule. `CSSStyleSheet#insertRule` method, and similar functions which might exist, which parse text into a single rule.
- "Parse a declaration" is used in '@supports' conditions. [CSS3-CONDITIONAL]
- "Parse a block's contents" is intended for parsing the contents of any block in CSS (including things like the style attribute), and APIs such as `the CSSStyleDeclaration cssText attribute`.
- "Parse a component value" is for things that need to consume a single value, like the parsing rules for 'attr()'.
- "Parse a list of component values" is for the contents of presentational attributes, which parse text into a single declaration's value, or for parsing a stand-alone selector [SELECT] or list of Media Queries [MEDIAQ], as in Selectors API or the `media` HTML attribute.

§ 5.4.1. Parse something according to a CSS grammar

It is often desirable to parse a string or token list to see if it matches some CSS grammar, and if it does, to destructure it according to the grammar. This section provides a generic hook for this kind of operation. It should be invoked like "parse *foo* as a CSS <color>", or similar.

This algorithm returns either failure, if the input does not match the provided grammar, or the result of parsing the input according to the grammar, which is an unspecified structure corresponding to the provided grammar specification. The return value must only be interacted with by specification prose, where the representation ambiguity is not problematic. If it is meant to be exposed outside of spec language, the spec using the result must explicitly translate it into a well-specified representation, such as, for example, by invoking a CSS serialization algorithm (like "serialize as a CSS <string> value").

> NOTE: This algorithm, and parse a comma-separated list according to a CSS grammar, are *usually* the only parsing algorithms other specs will want to call. The remaining parsing algorithms are meant mostly for [CSSOM] and related "explicitly constructing CSS structures" cases. Consult the CSSWG for guidance first if you think you need to use one of the other algorithms.

To ***parse something according to a CSS grammar*** (aka simply parse) given an *input* and a CSS *grammar* production:

1. Normalize *input*, and set *input* to the result.
2. Parse a list of component values from *input*, and let *result* be the return value.
3. Attempt to match *result* against *grammar*. If this is successful, return the matched result; otherwise, return failure.

§ 5.4.2. Parse a comma-separated list according to a CSS grammar

While one can definitely parse a value according to a grammar with commas in it, if *any* part of the value fails to parse, the entire thing doesn't parse, and returns failure.

Sometimes that's what's desired (such as in list-valued CSS properties); other times, it's better to let each comma-separated sub-part of the value parse separately, dealing with the parts that parse successfully one way, and the parts that fail to parse another way (typically ignoring them, such as in `<img sizes>`).

This algorithm provides an easy hook to accomplish exactly that. It returns a list of values split by "top-level" commas, where each values is either failure (if it failed to parse) or the result of parsing (an unspecified structure, as described in the parse algorithm).

To ***parse a comma-separated list according to a CSS grammar*** (aka parse a list) given an *input* and a CSS *grammar* production:

1. Normalize *input*, and set *input* to the result.
2. If *input* contains only <whitespace-token>s, return an empty list.
3. Parse a comma-separated list of component values from *input*, and let *list* be the return value.
4. For each *item* of *list*, replace *item* with the result of parsing *item* with *grammar*.
5. Return *list*.

§ 5.4.3. Parse a stylesheet

To ***parse a stylesheet*** from an *input* given an optional url *location*:

1. If *input* is a byte stream for a stylesheet, decode bytes from *input*, and set *input* to the result.
2. Normalize *input*, and set *input* to the result.
3. Create a new stylesheet, with its location set to *location* (or null, if *location* was not passed).
4. Consume a stylesheet's contents from *input*, and set the stylesheet's rules to the result.
5. Return the stylesheet.

§ 5.4.4. Parse a stylesheet's contents

To ***parse a stylesheet's contents*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Consume a stylesheet's contents from *input*, and return the result.

§ 5.4.5. Parse a block's contents

To ***parse a block's contents*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Consume a block's contents from *input*, and return the result.

§ 5.4.6. Parse a rule

To ***parse a rule*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Discard whitespace from *input*.
3. If the next token from *input* is an <EOF-token>, return a syntax error.

   Otherwise, if the next token from *input* is an <at-keyword-token>, consume an at-rule from *input*, and let *rule* be the return value.

   Otherwise, consume a qualified rule from *input* and let *rule* be the return value. If nothing was returned, return a syntax error.

4. Discard whitespace from *input*.
5. If the next token from *input* is an <EOF-token>, return *rule*. Otherwise, return a syntax error.

§ 5.4.7. Parse a declaration

> NOTE: Unlike "Parse a list of declarations", this parses only a declaration and not an at-rule.

To ***parse a declaration*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Discard whitespace from *input*.
3. Consume a declaration from *input*. If anything was returned, return it. Otherwise, return a syntax error.

§ 5.4.8. Parse a component value

To ***parse a component value*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Discard whitespace from *input*.
3. If *input* is empty, return a syntax error.
4. Consume a component value from *input* and let *value* be the return value.

5. Discard whitespace from *input*.
6. If *input* is empty, return *value*. Otherwise, return a syntax error.

§ 5.4.9. Parse a list of component values

To ***parse a list of component values*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Consume a list of component values from *input*, and return the result.

§ 5.4.10. Parse a comma-separated list of component values

To ***parse a comma-separated list of component values*** from *input*:

1. Normalize *input*, and set *input* to the result.
2. Let *groups* be an empty list.
3. While *input* is not empty:

    1. Consume a list of component values from *input*, with <comma-token> as the stop token, and append the result to *groups*.
    2. Discard a token from *input*.

4. Return *groups*.

§ 5.5. Parser Algorithms

The following algorithms comprise the parser. They are called by the parser entry points above, and generally should not be called directly by other specifications.

Note that CSS parsing is case-sensitive, and checking the validity of constructs in a given context must be done *during* parsing in at least some circumstances. This specification intentionally does not specify *how* sufficient context should be passed around to enable validity-checking.

§ 5.5.1. Consume a stylesheet's contents

To ***consume a stylesheet's contents*** from a token stream *input*:

Let *rules* be an initially empty list of rules.

Process *input*:

<whitespace-token>
: Discard a token from *input*.

<EOF-token>
: Return *rules*.

<CDO-token>
<CDC-token>
: Discard a token from *input*.

> ▶ **What's this for?**

<at-keyword-token>
: Consume an at-rule from *input*. If anything is returned, append it to *rules*.

anything else
: Consume a qualified rule from *input*. If anything is returned, append it to *rules*.

5.5.2. Consume an at-rule

To ***consume an at-rule*** from a token stream *input*, given an optional bool *nested* (default false):

Assert: The next token is an <at-keyword-token>.

Consume a token from *input*, and let *rule* be a new at-rule with its name set to the returned token's value, its prelude initially set to an empty list, and no declarations or child rules.

Process *input*:

<semicolon-token>
<EOF-token>
: Discard a token from *input*. If *rule* is valid in the current context, return it; otherwise return nothing.

<}-token>
: If *nested* is true:

- If *rule* is valid in the current context, return it.
- Otherwise, return nothing.

Otherwise, consume a token and append the result to *rule*'s prelude.

<{-token>
: Consume a block from *input*, and assign the results to *rule*'s lists of declarations and child rules.

If *rule* is valid in the current context, return it. Otherwise, return nothing.

anything else
: Consume a component value from *input* and append the returned value to *rule*'s prelude.

5.5.3. Consume a qualified rule

To ***consume a qualified rule***, from a token stream *input*, given an optional token *stop token* and an optional bool *nested* (default false):

Let *rule* be a new qualified rule with its prelude, declarations, and child rules all initially set to empty lists.

Process *input*:

<EOF-token>
***stop token* (if passed)**
: This is a parse error. Return nothing.

<}-token>
: This is a parse error. If *nested* is true, return nothing. Otherwise, consume a token and append the result to *rule*'s prelude.

<{-token>
: If the first two non-<whitespace-token> values of *rule*'s prelude are an <ident-token> whose value starts with "--" followed by a <colon-token>, then:

- If *nested* is true, consume the remnants of a bad declaration from *input*, with *nested* set to true, and return nothing.
- If *nested* is false, consume a block from *input*, and return nothing.

▶ **What's this check for?**

Otherwise, consume a block from *input*, and assign the results to *rule*'s lists of declarations and child rules.

If *rule* is valid in the current context, return it; otherwise return nothing.

anything else
Consume a component value from *input* and append the result to *rule*'s prelude.

§ 5.5.4. Consume a block

To ***consume a block***, from a token stream *input*:

Assert: The next token is a <{-token>.

Let *decls* be an empty list of declarations, and *rules* be an empty list of rules.

Discard a token from *input*. Consume a block's contents from *input* and assign the results to *decls* and *rules*. Discard a token from *input*.

Return *decls* and *rules*.

§ 5.5.5. Consume a block's contents

To ***consume a block's contents*** from a token stream *input*:

Let *decls* be an empty list of declarations, and *rules* be an empty list of rules.

Process *input*:

<whitespace-token>
<semicolon-token>
Discard a token from *input*.

<EOF-token>
<}-token>
Return *decls* and *rules*.

<at-keyword-token>
Consume an at-rule from *input*, with *nested* set to true. If a rule was returned, append it to *rules*.

anything else
Mark *input*.

Consume a declaration from *input*, with *nested* set to true. If a declaration was returned, append it to *decls*, and discard a mark from *input*.

Otherwise, restore a mark from *input*, then consume a qualified rule from *input*, with *nested* set to true, and <semicolon-token> as the *stop token*. If a rule was returned, append it to *rules*.

▶ **Implementation note**

§ 5.5.6. Consume a declaration

To ***consume a declaration*** from a token stream *input*, given an optional bool *nested* (default false):

Let *decl* be a new declaration, with an initially empty name and a value set to an empty list.

1. If the next token is an <ident-token>, consume a token from *input* and set *decl*'s name to the token's value.

   Otherwise, consume the remnants of a bad declaration from *input*, with *nested*, and return nothing.

2. Discard whitespace from *input*.
3. If the next token is a <colon-token>, discard a token from *input*.

   Otherwise, consume the remnants of a bad declaration from *input*, with *nested*, and return nothing.

4. Discard whitespace from *input*.
5. Consume a list of component values from *input*, with *nested*, and with <semicolon-token> as the stop token, and set *decl*'s value to the result.
6. If the last two non-<whitespace-token>s in *decl*'s value are a <delim-token> with the value "!" followed by an <ident-token> with a value that is an ASCII case-insensitive match for "important", remove them from *decl*'s value and set *decl*'s *important* flag.
7. While the last item in *decl*'s value is a <whitespace-token>, remove that token.
8. If *decl*'s name is a custom property name string, then set *decl*'s *original text* to the segment of the original source text string corresponding to the tokens of *decl*'s value.

   Otherwise, if *decl*'s value contains a top-level simple block with an associated token of <{-token>, and also contains *any other* non-<whitespace-token> value, return nothing. (That is, a top-level {}-block is only allowed as the entire value of a non-custom property.)

   Otherwise, if *decl*'s name is an ASCII case-insensitive match for "unicode-range", consume the value of a unicode-range descriptor from the segment of the original source text string corresponding to the tokens returned by the consume a list of component values call, and replace *decl*'s value with the result.

9. If *decl* is valid in the current context, return it; otherwise return nothing.

To ***consume the remnants of a bad declaration*** from a token stream *input*, given a bool *nested*:

Process *input*:

<eof-token>
<semicolon-token>
: Discard a token from *input*, and return nothing.

<}-token>
: If *nested* is true, return nothing. Otherwise, discard a token.

anything else
: Consume a component value from *input*, and do nothing.

§ 5.5.7. Consume a list of component values

To ***consume a list of component values*** from a token stream *input*, given an optional token *stop token* and an optional boolean *nested* (default false):

Let *values* be an empty list of component values.

Process *input*:

<eof-token>
***stop token* (if passed)**
: Return *values*.

<}-token>
: If *nested* is true, return *values*.

Otherwise, this is a parse error. Consume a token from *input* and append the result to *values*.

anything else
Consume a component value from *input*, and append the result to *values*.

§ 5.5.8. Consume a component value

To ***consume a component value*** from a token stream *input*:

Process *input*:

<{-token>
<[-token>
<(-token>
Consume a simple block from *input* and return the result.

<function-token>
Consume a function from *input* and return the result.

anything else
Consume a token from *input* and return the result.

§ 5.5.9. Consume a simple block

To ***consume a simple block*** from a token stream *input*:

Assert: the next token of *input* is <{-token>, <[-token>, or <(-token>.

Let *ending token* be the mirror variant of the next token. (E.g. if it was called with <[-token>, the *ending token* is <]-token>.)

Let *block* be a new simple block with its associated token set to the next token and with its value initially set to an empty list.

Discard a token from *input*.

Process *input*:

<eof-token>
ending token
Discard a token from *input*. Return *block*.

anything else
Consume a component value from *input* and append the result to *block*'s value.

§ 5.5.10. Consume a function

To ***consume a function*** from a token stream *input*:

Assert: The next token is a <function-token>.

Consume a token from *input*, and let *function* be a new function with its name equal the returned token's value, and a value set to an empty list.

Process *input*:

<eof-token>
<)-token>
Discard a token from *input*. Return *function*.

anything else
Consume a component value from *input* and append the result to *function*'s value.

§ 5.5.11. Consume a 'unicode-range' value

To ***consume the value of a unicode-range descriptor*** from a string *input string*:

1. Let *tokens* be the result of tokenizing *input string* with *unicode ranges allowed* set to true. Let *input* be a new token stream from *tokens*.
2. Consume a list of component values from *input*, and return the result.

NOTE: The existence of this algorithm is due to a design mistake in early CSS. It should never be reproduced.

§ 6. The *An*+*B* microsyntax

Several things in CSS, such as the ':nth-child()' pseudoclass, need to indicate indexes in a list. The *An*+*B* microsyntax is useful for this, allowing an author to easily indicate single elements or all elements at regularly-spaced intervals in a list.

The ***An*+*B*** notation defines an integer step (*A*) and offset (*B*), and represents the *An*+*B*th elements in a list, for every positive integer or zero value of *n*, with the first element in the list having index 1 (not 0).

For values of *A* and *B* greater than 0, this effectively divides the list into groups of *A* elements (the last group taking the remainder), and selecting the *B*th element of each group.

The *An*+*B* notation also accepts the 'even' and 'odd' keywords, which have the same meaning as '2n' and '2n+1', respectively.

EXAMPLE 4

Examples:

```
2n+0   /* represents all of the even elements in the list */

even   /* same */

4n+1   /* represents the 1st, 5th, 9th, 13th, etc. elements in the list */
```

The values of *A* and *B* can be negative, but only the positive results of *An*+*B*, for $n \geq 0$, are used.

EXAMPLE 5

Example:

```
-1n+6   /* represents the first 6 elements of the list */

-4n+10  /* represents the 2nd, 6th, and 10th elements of the list */
```

If both *A* and *B* are 0, the pseudo-class represents no element in the list.

§ 6.1. Informal Syntax Description

This section is non-normative.

When *A* is 0, the *An* part may be omitted (unless the *B* part is already omitted). When *An* is not included and *B* is non-negative, the '+' sign before *B* (when allowed) may also be omitted. In this case the syntax simplifies to just *B*.

EXAMPLE 6

Examples:

```
0n+5   /* represents the 5th element in the list */

5      /* same */
```

When *A* is 1 or -1, the `1` may be omitted from the rule.

EXAMPLE 7

Examples:

The following notations are therefore equivalent:

```
1n+0   /* represents all elements in the list */

n+0    /* same */

n      /* same */
```

If *B* is 0, then every *A*th element is picked. In such a case, the +*B* (or -*B*) part may be omitted unless the *A* part is already omitted.

EXAMPLE 8

Examples:

```
2n+0   /* represents every even element in the list */

2n     /* same */
```

When B is negative, its minus sign replaces the '+' sign.

EXAMPLE 9

Valid example:

```
3n-6
```

Invalid example:

```
3n + -6
```

Whitespace is permitted on either side of the '+' or '-' that separates the *An* and *B* parts when both are present.

EXAMPLE 10 ¶

Valid Examples with white space:

```
3n + 1

+3n - 2

-n+ 6

+6
```

Invalid Examples with white space:

```
3 n

+ 2n

+ 2
```

§ 6.2. The <an+b> type

The *An+B* notation was originally defined using a slightly different tokenizer than the rest of CSS, resulting in a somewhat odd definition when expressed in terms of CSS tokens. This section describes how to recognize the *An+B* notation in terms of CSS tokens (thus defining the <*an+b*> type for CSS grammar purposes), and how to interpret the CSS tokens to obtain values for *A* and *B*.

The <*an+b*> type is defined (using the Value Definition Syntax in the Values & Units spec) as:

```
<an+b> =
  odd | even |
  <integer> |

  <n-dimension> |
  '+'?† n |
  -n |

  <ndashdigit-dimension> |
  '+'?† <ndashdigit-ident> |
  <dashndashdigit-ident> |

  <n-dimension> <signed-integer> |
  '+'?† n <signed-integer> |
  -n <signed-integer> |

  <ndash-dimension> <signless-integer> |
  '+'?† n- <signless-integer> |
  -n- <signless-integer> |

  <n-dimension> ['+' | '-'] <signless-integer> |
  '+'?† n ['+' | '-'] <signless-integer> |
  -n ['+' | '-'] <signless-integer>
```

where:

- ***'<n-dimension>'*** is a <dimension-token> with its type flag set to "integer", and a unit that is an ASCII case-insensitive match for "n"
- ***'<ndash-dimension>'*** is a <dimension-token> with its type flag set to "integer", and a unit that is an ASCII case-insensitive match for "n-"
- ***'<ndashdigit-dimension>'*** is a <dimension-token> with its type flag set to "integer", and a unit that is an ASCII case-insensitive match for "n-*", where "*" is a series of one or more digits
- ***'<ndashdigit-ident>'*** is an <ident-token> whose value is an ASCII case-insensitive match for "n-*", where "*" is a series of one or more digits
- ***'<dashndashdigit-ident>'*** is an <ident-token> whose value is an ASCII case-insensitive match for "-n-*", where "*" is a series of one or more digits
- ***'<integer>'*** is a <number-token> with its type flag set to "integer"
- ***'<signed-integer>'*** is a <number-token> with its type flag set to "integer", and a sign character
- ***'<signless-integer>'*** is a <number-token> with its type flag set to "integer", and no sign character

†: When a plus sign (+) precedes an ident starting with "n", as in the cases marked above, there must be no whitespace between the two tokens, or else the tokens do not match the above grammar. Whitespace is valid (and ignored) between any other two tokens.

The clauses of the production are interpreted as follows:

'odd'
: *A* is 2, *B* is 1.

'even'
: *A* is 2, *B* is 0.

<integer>
: *A* is 0, *B* is the integer's value.

<n-dimension>
'+'? n
-n
: *A* is the dimension's value, 1, or -1, respectively. *B* is 0.

<ndashdigit-dimension>
'+'? *<ndashdigit-ident>*
: *A* is the dimension's value or 1, respectively. *B* is the dimension's unit or ident's value, respectively, with the first code point removed and the remainder interpreted as a base-10 number. B is negative.

<dashndashdigit-ident>
: *A* is -1. *B* is the ident's value, with the first two code points removed and the remainder interpreted as a base-10 number. B is negative.

<n-dimension> <signed-integer>
'+'? n *<signed-integer>*
-n *<signed-integer>*
: *A* is the dimension's value, 1, or -1, respectively. *B* is the integer's value.

<ndash-dimension> <signless-integer>
'+'? n- *<signless-integer>*
-n- *<signless-integer>*
: *A* is the dimension's value, 1, or -1, respectively. *B* is the negation of the integer's value.

<n-dimension>* ['+' | '-'] *<signless-integer>
'+'? n ['+' | '-'] *<signless-integer>*
-n ['+' | '-'] *<signless-integer>*
: *A* is the dimension's value, 1, or -1, respectively. *B* is the integer's value. If a `'-'` was provided between the two, *B* is instead the negation of the integer's value.

§ 7. Defining Grammars for Rules and Other Values

CSS Values 4 § 2 Value Definition Syntax defines how to specify a grammar for properties. This section extends those definitions to also allow specifying a grammar for rules.

Non-terminals representing the entire grammar of an at-rule are written as an @ character followed by the at-rule's name, between '<' and '>', e.g. <@media> to represent the '@media' rule.

The bracketed range notation can be used on any of the numeric token non-terminals.

Several types of tokens are written literally, without quotes:

- <ident-token>s (such as `auto`, `disc`, etc), which are simply written as their value.
- <at-keyword-token>s, which are written as an @ character followed by the token's value, like `@media`.
- <function-token>s, which are written as the function name followed by a (character, like `translate(`.
- The <colon-token> (written as `:`), <comma-token> (written as `,`), <semicolon-token> (written as `;`), <(-token>, <)-token>, <{-token>, and <}-token>s.

Tokens match if their value is a match for the value defined in the grammar. Unless otherwise specified, all matches are ASCII case-insensitive.

> NOTE: Although it is possible, with escaping, to construct an <ident-token> whose value ends with `(` or starts with `@`, such a tokens is not a <function-token> or an <at-keyword-token> and does not match corresponding grammar definitions.

<delim-token>s are written with their value enclosed in single quotes. For example, a <delim-token> containing the "+" code point is written as `'+'`. Similarly, the <[-token> and <]-token>s must be written in single quotes, as they're used by the syntax of the grammar itself to group clauses. <whitespace-token> is never indicated in the grammar; <whitespace-token>s are allowed before, after, and between any two tokens, unless explicitly specified otherwise in prose definitions. (For example, if the prelude of a rule is a selector, whitespace is significant.)

When defining a function or a block, the ending token must be specified in the grammar, but if it's not present in the eventual token stream, it still matches.

> EXAMPLE 11
> For example, the syntax of the 'translateX()' function is:
>
> ```
> translateX( <translation-value> )
> ```
>
> However, the stylesheet may end with the function unclosed, like:
>
> ```
> .foo { transform: translate(50px
> ```
>
> The CSS parser parses this as a style rule containing one declaration, whose value is a function named "translate". This matches the above grammar, even though the ending token didn't appear in the token stream, because by the time the parser is finished, the presence of the ending token is no longer possible to determine; all you have is the fact that there's a block and a function.

§ 7.1. Defining Block Contents: the <block-contents>, <declaration-list>, <qualified-rule-list>, <declaration-rule-list>, and <rule-list> productions

The CSS parser is agnostic as to the contents of blocks—they're all parsed with the same algorithm, and differentiate themselves solely by what constructs are valid.

When writing a rule grammar, ***'<block-contents>'*** represents this agnostic parsing. It must only be used as the sole value in a block, and represents that no restrictions are implicitly placed on what the block can contain.

Accompanying prose must define what is valid and invalid in this context. If any declarations are valid, and are property declarations, it must define whether they interact with the cascade; if they do, it must define their specificity and how they use `!important`.

In many cases, however, a block can't validly contain *any* constructs of a given type. To represent these cases more explicitly, the following productions may be used

- ***'<declaration-list>'***: only declarations are allowed; at-rules and qualified rules are automatically invalid.
- ***'<qualified-rule-list>'***: only qualified rules are allowed; declarations and at-rules are automatically invalid.
- ***'<at-rule-list>'***: only at-rules are allowed; declarations and qualified rules are automatically invalid.
- ***'<declaration-rule-list>'***: declarations and at-rules are allowed; qualified rules are automatically invalid.
- ***'<rule-list>'***: qualified rules and at-rules are allowed; declarations are automatically invalid.

All of these are exactly equivalent to <block-contents> in terms of parsing, but the accompanying prose only has to define validity for the categories that aren't automatically invalid.

EXAMPLE 12

Some examples of the various productions:

- A top-level '@media' uses <rule-list> for its block, while a nested one [CSS-NESTING-1] uses <block-contents>.
- Style rules use <block-contents>.
- '@font-face' uses <declaration-list>.
- '@page' uses <declaration-rule-list>.
- '@keyframes' uses <qualified-rule-list>

EXAMPLE 13

For example, the grammar for '@font-face' can be written as:

```
<@font-face> = @font-face { <declaration-list> }
```

and then accompanying prose defines the valid descriptors allowed in the block.

The grammar for '@keyframes' can be written as:

```
<@keyframes> = @keyframes { <qualified-rule-list> }
<keyframe-rule> = <keyframe-selector> { <declaration-list> }
```

and then accompanying prose defines that only <keyframe-rule>s are allowed in '@keyframes', and that <keyframe-rule>s accept all animatable CSS properties, plus the 'animation-timing-function' property, but they do not interact with the cascade.

§ 7.2. Defining Arbitrary Contents: the <declaration-value> and <any-value> productions

In some grammars, it is useful to accept any reasonable input in the grammar, and do more specific error-handling on the contents manually (rather than simply invalidating the construct, as grammar mismatches tend to do).

For example, custom properties allow any reasonable value, as they can contain arbitrary pieces of other CSS properties, or be used for things that aren't part of existing CSS at all. For another example, the <general-enclosed> production in Media

Queries defines the bounds of what future syntax MQs will allow, and uses special logic to deal with "unknown" values.

To aid in this, two additional productions are defined:

The ***'<declaration-value>'*** production matches *any* sequence of one or more tokens, so long as the sequence does not contain <bad-string-token>, <bad-url-token>, unmatched <)-token>, <]-token>, or <}-token>, or top-level <semicolon-token> tokens or <delim-token> tokens with a value of "!". It represents the entirety of what a valid declaration can have as its value.

The ***'<any-value>'*** production is identical to <declaration-value>, but also allows top-level <semicolon-token> tokens and <delim-token> tokens with a value of "!". It represents the entirety of what valid CSS can be in any context.

§ 8. CSS stylesheets

To ***parse a CSS stylesheet***, first parse a stylesheet. Interpret all of the resulting top-level qualified rules as style rules, defined below.

If any style rule is invalid, or any at-rule is not recognized or is invalid according to its grammar or context, it's a parse error. Discard that rule.

§ 8.1. Style rules

A ***style rule*** is a qualified rule that associates a selector list with a list of property declarations and possibly a list of nested rules. They are also called rule sets in [CSS2]. CSS Cascading and Inheritance [CSS-CASCADE-3] defines how the declarations inside of style rules participate in the cascade.

The prelude of the qualified rule is parsed as a <selector-list>. If this returns failure, the entire style rule is invalid.

The content of the qualified rule's block is parsed as a <declaration-list>. Qualified rules in this block are also style rules. Unless defined otherwise by another specification or a future level of this specification, at-rules in that list are invalid and must be ignored.

> NOTE: [CSS-NESTING-1] defines that conditional group rules and some other at-rules are allowed inside of style rules.

Declarations for an unknown CSS property or whose value does not match the syntax defined by the property are invalid and must be ignored. The validity of the style rule's contents have no effect on the validity of the style rule itself. Unless otherwise specified, property names are ASCII case-insensitive.

> NOTE: The names of Custom Properties [CSS-VARIABLES] are case-sensitive.

Qualified rules at the top-level of a CSS stylesheet are style rules. Qualified rules in other contexts may or may not be style rules, as defined by the context.

> EXAMPLE 14 ¶
> For example, qualified rules inside '@media' rules [CSS3-CONDITIONAL] are style rules, but qualified rules inside '@keyframes' rules [CSS3-ANIMATIONS] are not.

§ 8.2. At-rules

An ***at-rule*** is a rule that begins with an at-keyword, and can thus be distinguished from style rules in the same context.

At-rules are used to:

- group and structure style rules and other at-rules such as in conditional group rules
- declare style information that is not associated with a particular element, such as defining counter styles
- manage syntactic constructs such as imports and namespaces keyword mappings
- and serve other miscellaneous purposes not served by a style rule

At-rules take many forms, depending on the specific rule and its purpose, but broadly speaking there are two kinds: ***statement at-rules*** which are simpler constructs that end in a semicolon, and ***block at-rules*** which end in a {}-block that can contain nested qualified rules, at-rules, or declarations.

Block at-rules will typically contain a collection of (generic or at-rule–specific) at-rules, qualified rules, and/or descriptor declarations subject to limitations defined by the at-rule. ***Descriptors*** are similar to properties (and are declared with the same syntax) but are associated with a particular type of at-rule rather than with elements and boxes in the tree.

§ 8.3. The '@charset' Rule

The algorithm used to determine the fallback encoding for a stylesheet looks for a specific byte sequence as the very first few bytes in the file, which has the syntactic form of an at-rule named "@charset".

However, there is no actual at-rule named ***'@charset'***. When a stylesheet is actually parsed, any occurrences of an '@charset' rule must be treated as an unrecognized rule, and thus dropped as invalid when the stylesheet is grammar-checked.

> NOTE: In CSS 2.1, '@charset' was a valid rule. Some legacy specs may still refer to a '@charset' rule, and explicitly talk about its presence in the stylesheet.

§ 9. Serialization

The tokenizer described in this specification does not produce tokens for comments, or otherwise preserve them in any way. Implementations may preserve the contents of comments and their location in the token stream. If they do, this preserved information must have no effect on the parsing step.

This specification does not define how to serialize CSS in general, leaving that task to the [CSSOM] and individual feature specifications. In particular, the serialization of comments and whitespace is not defined.

The only requirement for serialization is that it must "round-trip" with parsing, that is, parsing the stylesheet must produce the same data structures as parsing, serializing, and parsing again, except for consecutive <whitespace-token>s, which may be collapsed into a single token.

> NOTE: This exception can exist because CSS grammars always interpret any amount of whitespace as identical to a single space.

To satisfy this requirement:

- A <delim-token> containing U+005C REVERSE SOLIDUS (\) must be serialized as U+005C REVERSE SOLIDUS followed by a newline. (The tokenizer only ever emits such a token followed by a <whitespace-token> that starts with a newline.)
- A <hash-token> with the "unrestricted" type flag may not need as much escaping as the same token with the "id" type flag.
- The unit of a <dimension-token> may need escaping to disambiguate with scientific notation.
- For any consecutive pair of tokens, if the first token shows up in the row headings of the following table, and the second token shows up in the column headings, and there's a ✗ in the cell denoted by the intersection of the chosen row and column, the pair of tokens must be serialized with a comment between them.
  If the tokenizer preserves comments, and there were comments originally between the token pair, the preserved comment(s) should be used; otherwise, an empty comment (`/**/`) must be inserted. (Preserved comments may be reinserted even if the following tables don't require a comment between two tokens.)

  Single characters in the row and column headings represent a <delim-token> with that value, except for "(", which represents a (-token.

	ident	function	url	bad url	-	number	percentage	dimension	CDC	(	*	%
ident	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
at-keyword	✗	✗	✗	✗	✗	✗	✗	✗	✗			
hash	✗	✗	✗	✗	✗	✗	✗	✗	✗			
dimension	✗	✗	✗	✗	✗	✗	✗	✗	✗			
#	✗	✗	✗	✗	✗	✗	✗	✗	✗			
-	✗	✗	✗	✗	✗	✗	✗	✗	✗			
number	✗	✗	✗	✗		✗	✗	✗	✗			✗
@	✗	✗	✗	✗	✗				✗			
.						✗	✗	✗				
+						✗	✗	✗				
/											✗	

§ 9.1. Serializing <*an+b*>

To ***serialize an <an+b> value***, with integer values *A* and *B*:

1. If *A* is zero, return the serialization of *B*.
2. Otherwise, let *result* initially be an empty string.

3↦ ***A* is `1`**
Append "n" to *result*.

↪ ***A* is `-1`**
Append "-n" to *result*.

↪ ***A* is non-zero**
Serialize *A* and append it to *result*, then append "n" to *result*.

7↦ ***B* is greater than zero**
Append "+" to *result*, then append the serialization of *B* to *result*.

↪ ***B* is less than zero**

Append the serialization of *B* to *result*.

10. Return *result*.

§ 10. Privacy Considerations

This specification introduces no new privacy concerns.

§ 11. Security Considerations

This specification improves security, in that CSS parsing is now unambiguously defined for all inputs.

Insofar as old parsers, such as whitelists/filters, parse differently from this specification, they are somewhat insecure, but the previous parsing specification left a lot of ambiguous corner cases which browsers interpreted differently, so those filters were potentially insecure already, and this specification does not worsen the situation.

§ 12. Changes

This section is non-normative.

§ 12.1. Changes from the 24 December 2021 Candidate Recommendation Draft

The following substantive changes were made:

- The definition of non-ASCII code point was restricted to omit some potentially troublesome codepoints. (7129)
- Defined the `<foo()>` and `<@foo>` productions. (5728)
- Allow nested style rules. (7961).
- As part of allowing Nesting, significantly rewrote the entire parsing section. Notably, removed "parse a list of rules" and "parse a list of declarations" in favor of "parse a stylesheet's contents" and "parse a block's contents". Only additional normative change is that semicolons trigger slightly different error-recovery now in some contexts, so that parsing of things like '@media' blocks is identical whether they're nested or not. (8834).
- Removed the attempt at a `<urange>` production in terms of existing tokens, instead relying on a special re-parse from source specifically when you're parsing the 'unicode-range' descriptor. (8835)
- Since the above removed the main need to preserve a token's "representation" (the original string it was parsed from), removed the rest of the references to "representation" in the An+B section and instead just recorded the few bits of information necessary for that (whether or not the number had an explicit sign).
- Explicitly noted that some uses (custom properties, the 'unicode-range' descriptor) require access to the original string of a declaration's entire value, and marked where that occurs in the parser.

§ 12.2. Changes from the 16 August 2019 Candidate Recommendation

The following substantive changes were made:

- Added a new § 5.4.2 Parse a comma-separated list according to a CSS grammar algorithm.
- Added a new "Parse a style block's content" algorithm and a corresponding <style-block> production, and defined that style rules use it.
- Aligned parse a stylesheet with the Fetch-related shenanigans. (See commit.)

> To parse a stylesheet from an *input* given an optional url *location* :
>
> 1. ...
> 2. Create a new stylesheet , with its location set to *location* (or null, if *location* was not passed).
> 3. ...

The following editorial changes were made:

- Added § 8.2 At-rules to provide definitions for at-rules, statement at-rules, block at-rules, and descriptors. (5633)
- Improved the definition text for declaration, and added definitions for property declarations and descriptor declarations.
- Switched to consistently refer to ident sequence, rather than sometimes using the term “name”.
- Explicitly named several of the pre-tokenizing processes, and explicitly referred to them in the parsing entry points (rather than relying on a blanket "do X at the start of these algorithms" statement).
- Added more entries to the "put a comment between them" table, to properly handle the fact that idents can now start with `--`. (6874)

§ 12.3. Changes from the 20 February 2014 Candidate Recommendation

The following substantive changes were made:

- Removed <unicode-range-token>, in favor of creating a <urange> production.
- url() functions that contain a string are now parsed as normal <function-token>s. url() functions that contain "raw" URLs are still specially parsed as <url-token>s.
- Fixed a bug in the "Consume a URL token" algorithm, where it didn’t consume the quote character starting a string before attempting to consume the string.
- Fixed a bug in several of the parser algorithms related to the current/next token and things getting consumed early/late.
- Fix several bugs in the tokenization and parsing algorithms.
- Change the definition of ident-like tokens to allow "--" to start an ident. As part of this, rearrange the ordering of the clauses in the "-" step of consume a token so that <CDC-token>s are recognized as such instead of becoming a ‘--’ <ident-token>.
- Don’t serialize the digit in an <an+b> when A is 1 or -1.
- Define all tokens to have a representation.
- Fixed minor bug in check if two code points are a valid escape—a \ followed by an EOF is now correctly reported as *not* a valid escape. A final \ in a stylesheet now just emits itself as a <delim-token>.
- @charset is no longer a valid CSS rule (there’s just an encoding declaration that *looks* like a rule named @charset)
- Trimmed whitespace from the beginning/ending of a declaration’s value during parsing.
- Removed the Selectors-specific tokens, per WG resolution.
- Filtered surrogates from the input stream, per WG resolution. Now the entire specification operates only on scalar values.

The following editorial changes were made:

- The "Consume a string token" algorithm was changed to allow calling it without specifying an explicit ending token, so that it uses the current token instead. The three call-sites of the algorithm were changed to use that form.
- Minor editorial restructuring of algorithms.
- Added the parse and parse a comma-separated list of component values API entry points.

- Added the <declaration-value> and <any-value> productions.
- Removed "code point" and "surrogate code point" in favor of the identical definitions in the Infra Standard.
- Clarified on every range that they are inclusive.
- Added a column to the comment-insertion table to handle a number token appearing next to a "%" delim token.

A Disposition of Comments is available.

§ 12.4. Changes from the 5 November 2013 Last Call Working Draft

- The Serialization section has been rewritten to make only the "round-trip" requirement normative, and move the details of how to achieve it into a note. Some corner cases in these details have been fixed.
- [ENCODING] has been added to the list of normative references. It was already referenced in normative text before, just not listed as such.
- In the algorithm to determine the fallback encoding of a stylesheet, limit the `@charset` byte sequence to 1024 bytes. This aligns with what HTML does for `<meta charset>` and makes sure the size of the sequence is bounded. This only makes a difference with leading or trailing whitespace in the encoding label:

  ```
  @charset "   (lots of whitespace)   utf-8";
  ```

§ 12.5. Changes from the 19 September 2013 Working Draft

- The concept of environment encoding was added. The behavior does not change, but some of the definitions should be moved to the relevant specs.

§ 12.6. Changes from CSS 2.1 and Selectors Level 3

> NOTE: The point of this spec is to match reality; changes from CSS2.1 are nearly always because CSS 2.1 specified something that doesn't match actual browser behavior, or left something unspecified. If some detail doesn't match browsers, please let me know as it's almost certainly unintentional.

Changes in decoding from a byte stream:

- Only detect '@charset' rules in ASCII-compatible byte patterns.
- Ignore '@charset' rules that specify an ASCII-incompatible encoding, as that would cause the rule itself to not decode properly.
- Refer to [ENCODING] rather than the IANA registry for character encodings.

Tokenization changes:

- Any U+0000 NULL code point in the CSS source is replaced with U+FFFD REPLACEMENT CHARACTER.
- Any hexadecimal escape sequence such as '\0' that evaluates to zero produce U+FFFD REPLACEMENT CHARACTER rather than U+0000 NULL.
- The definition of non-ASCII ident code point was changed to be consistent with HTML's valid custom element names.
- Tokenization does not emit COMMENT or BAD_COMMENT tokens anymore. BAD_COMMENT is now considered the same as a normal token (not an error). Serialization is responsible for inserting comments as necessary between tokens that need to be separated, e.g. two consecutive <ident-token>s.
- The <unicode-range-token> was removed, as it was low value and occasionally actively harmful. ('u+a { font-weight: bold; }' was an invalid selector, for example...)

Instead, a <urange> production was added, based on token patterns. It is technically looser than what 2.1 allowed (any number of digits and ? characters), but not in any way that should impact its use in practice.

- Apply the EOF error handling rule in the tokenizer and emit normal <string-token> and <url-token> rather than BAD_STRING or BAD_URI on EOF.
- The BAD_URI token (now <bad-url-token>) is "self-contained". In other words, once the tokenizer realizes it's in a <bad-url-token> rather than a <url-token>, it just seeks forward to look for the closing), ignoring everything else. This behavior is simpler than treating it like a <function-token> and paying attention to opened blocks and such. Only WebKit exhibits this behavior, but it doesn't appear that we've gotten any compat bugs from it.
- The <comma-token> has been added.
- <number-token>, <percentage-token>, and <dimension-token> have been changed to include the preceding +/- sign as part of their value (rather than as a separate <delim-token> that needs to be manually handled every time the token is mentioned in other specs). The only consequence of this is that comments can no longer be inserted between the sign and the number.
- Scientific notation is supported for numbers/percentages/dimensions to match SVG, per WG resolution.
- Hexadecimal escape for surrogate now emit a replacement character rather than the surrogate. This allows implementations to safely use UTF-16 internally.

Parsing changes:

- Any list of declarations now also accepts at-rules, like '@page', per WG resolution. This makes a difference in error handling even if no such at-rules are defined yet: an at-rule, valid or not, ends at a {} block without a <semicolon-token> and lets the next declaration begin.
- The handling of some miscellaneous "special" tokens (like an unmatched <}-token>) showing up in various places in the grammar has been specified with some reasonable behavior shown by at least one browser. Previously, stylesheets with those tokens in those places just didn't match the stylesheet grammar at all, so their handling was totally undefined. Specifically:
  - [] blocks, () blocks and functions can now contain {} blocks, <at-keyword-token>s or <semicolon-token>s
  - Qualified rule preludes can now contain semicolons
  - Qualified rule and at-rule preludes can now contain <at-keyword-token>s

An+B changes from Selectors Level 3 [SELECT]:

- The *An+B* microsyntax has now been formally defined in terms of CSS tokens, rather than with a separate tokenizer. This has resulted in minor differences:
  - In some cases, minus signs or digits can be escaped (when they appear as part of the unit of a <dimension-token> or <ident-token>).

§ Acknowledgments

Thanks for feedback and contributions from Anne van Kesteren, David Baron, Elika J. Etemad (fantasai), Henri Sivonen, Johannes Koch, 呂康豪 (Kang-Hao Lu), Marc O'Morain, Raffaello Giulietti, Simon Pieter, Tyler Karaszewski, and Zack Weinberg.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT",

“RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 15

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

UAs MUST provide an accessible alternative.

▼ TESTS

Tests relating to the content of this specification may be documented in “Tests” blocks like this one. Any such block is non-normative.

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

renderer
: A UA that interprets the semantics of a style sheet and renders documents that use them.

authoring tool
: A UA that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and ignore as appropriate) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends following best practices for the implementation of unstable features and proprietary extensions to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group's website at http://www.w3.org/Style/CSS/Test/. Questions should be directed to the public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

<{-token>, in § 4

<}-token>, in § 4

tokenization, in § 4

tokenize, in § 4

tokens, in § 5.3

token stream, in § 5.3

<unicode-range-token>, in § 4

uppercase letter, in § 4.2

<url-token>, in § 4

whitespace, in § 4.2

<whitespace-token>, in § 4

would start an ident sequence, in § 4.3.9

would start a number, in § 4.3.10

would start a unicode-range, in § 4.3.11

§ Terms defined by reference

[CSS-CASCADE-5] defines the following terms:

- @import
- property

[CSS-COLOR-4] defines the following terms:

- blue
- color

[CSS-COLOR-5] defines the following terms:

- <color>

[CSS-COUNTER-STYLES-3] defines the following terms:

- counter style

[CSS-FONTS-4] defines the following terms:

- font
- unicode-range

[CSS-FONTS-5] defines the following terms:

- @font-face

[CSS-PAGE-3] defines the following terms:

- :left
- @page

[CSS-SYNTAX-3] defines the following terms:

- <style-block>
- <urange>
- non-ascii code point
- parse a list of declarations

[CSS-TEXT-DECOR-3] defines the following terms:

- text-decoration

[CSS-TEXT-DECOR-4] defines the following terms:

- underline

[CSS-TRANSFORMS-1] defines the following terms:

- translatex()

[CSS-TYPED-OM-1] defines the following terms:

- custom property name string

[CSS-VALUES-4] defines the following terms:

- +
- <string>
- ?
- url()
- |

[CSS-VALUES-5] defines the following terms:

- attr()

[CSS-VARIABLES-2] defines the following terms:

- custom property
- var()

[CSS3-ANIMATIONS] defines the following terms:

- <keyframe-selector>
- @keyframes
- animation-timing-function

[CSS3-CONDITIONAL] defines the following terms:

- @media
- @supports
- conditional group rule

[CSSOM] defines the following terms:

- cssText
- insertRule(rule)
- location
- replace(text)

[ENCODING] defines the following terms:

- decode
- get an encoding

[HTML] defines the following terms:

- a
- p
- script
- sizes
- style
- valid custom element name

[INFRA] defines the following terms:

- ascii case-insensitive
- code point
- for each
- item
- list
- pop
- remove
- scalar value
- stack
- string
- struct
- surrogate

[MEDIAQUERIES-5] defines the following terms:

- <general-enclosed>

[SELECTORS-4] defines the following terms:

- +
- :nth-child()
- <selector-list>
- selector list

[URL] defines the following terms:

- url

§ References

§ Normative References

[CSS-CASCADE-3]
Elika Etemad; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 3*. URL: https://drafts.csswg.org/css-cascade-3/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. URL: https://drafts.csswg.org/css-cascade-5/

[CSS-COUNTER-STYLES-3]
Tab Atkins Jr.. *CSS Counter Styles Level 3*. URL: https://drafts.csswg.org/css-counter-styles/

[CSS-FONTS-4]
John Daggett; Myles Maxfield; Chris Lilley. *CSS Fonts Module Level 4*. URL: https://drafts.csswg.org/css-fonts-4/

[CSS-PAGE-3]
Elika Etemad. *CSS Paged Media Module Level 3*. URL: https://drafts.csswg.org/css-page-3/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. URL: https://drafts.csswg.org/css-syntax/

[CSS-TYPED-OM-1]
Shane Stephens; Tab Atkins Jr.; Naina Raisinghani. *CSS Typed OM Level 1*. URL: https://drafts.css-houdini.org/css-typed-om-1/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. URL: https://drafts.csswg.org/css-values-4/

[CSS-VARIABLES-2]
CSS Custom Properties for Cascading Variables Module Level 2. Editor's Draft. URL: https://drafts.csswg.org/css-variables-2/

[CSS3-CONDITIONAL]
David Baron; Elika Etemad; Chris Lilley. *CSS Conditional Rules Module Level 3*. URL: https://drafts.csswg.org/css-conditional-3/

[CSSOM]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. URL: https://drafts.csswg.org/cssom/

[ENCODING]
Anne van Kesteren. *Encoding Standard*. Living Standard. URL: https://encoding.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[SELECTORS-4]
Elika Etemad; Tab Atkins Jr.. *Selectors Level 4*. URL: https://drafts.csswg.org/selectors/

[URL]
Anne van Kesteren. *URL Standard*. Living Standard. URL: https://url.spec.whatwg.org/

§ Informative References

[CSS-COLOR-4]
Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. URL: https://drafts.csswg.org/css-color/

[CSS-COLOR-5]
Chris Lilley; et al. *CSS Color Module Level 5*. URL: https://drafts.csswg.org/css-color-5/

[CSS-FONTS-5]
Myles Maxfield; Chris Lilley. *CSS Fonts Module Level 5*. URL: https://drafts.csswg.org/css-fonts-5/

[CSS-NAMESPACES-3]
Elika Etemad. *CSS Namespaces Module Level 3*. URL: https://drafts.csswg.org/css-namespaces/

[CSS-NESTING-1]
Tab Atkins Jr.; Adam Argyle. *CSS Nesting Module*. URL: https://drafts.csswg.org/css-nesting/

[CSS-TEXT-DECOR-3]
Elika Etemad; Koji Ishii. *CSS Text Decoration Module Level 3*. URL: https://drafts.csswg.org/css-text-decor-3/

[CSS-TEXT-DECOR-4]
Elika Etemad; Koji Ishii. *CSS Text Decoration Module Level 4*. URL: https://drafts.csswg.org/css-text-decor-4/

[CSS-TRANSFORMS-1]
Simon Fraser; et al. *CSS Transforms Module Level 1*. URL: https://drafts.csswg.org/css-transforms/

[CSS-VALUES-5]
CSS Values and Units Module Level 5. Editor's Draft. URL: https://drafts.csswg.org/css-values-5/

[CSS-VARIABLES]
Tab Atkins Jr.. *CSS Custom Properties for Cascading Variables Module Level 1*. URL: https://drafts.csswg.org/css-variables/

[CSS2]
Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. URL: https://drafts.csswg.org/css2/

[CSS3-ANIMATIONS]
David Baron; et al. *CSS Animations Level 1*. URL: https://drafts.csswg.org/css-animations/

[MEDIAQ]
Florian Rivoal; Tab Atkins Jr.. *Media Queries Level 4*. URL: https://drafts.csswg.org/mediaqueries-4/

[MEDIAQUERIES-5]
Dean Jackson; et al. *Media Queries Level 5*. URL: https://drafts.csswg.org/mediaqueries-5/

[SELECT]
Tantek Çelik; et al. *Selectors Level 3*. URL: https://drafts.csswg.org/selectors-3/