W3C Candidate Recommendation

CSS Syntax Module Level 3

W3C Candidate Recommendation, 20 February 2014

This version:
http://www.w3.org/TR/2014/CR-css-syntax-3-20140220/

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

Editor's Draft:
http://dev.w3.org/csswg/css-syntax/

Previous Version:
http://www.w3.org/TR/2013/WD-css-syntax-3-20131105/

Feedback:
www-style@w3.org with subject line "`[css-syntax]` … *`message topic`* …"(archives)

Test Suite:
None Yet

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

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, in speech, etc.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document was produced by the CSS Working Group as a Candidate Recommendation.

A Candidate Recommendation is a document that has been widely reviewed and is ready for implementation. W3C encourages everybody to implement this specification and return comments to the (archived) public mailing list www-style@w3.org (see instructions). When sending e-mail, please put the text “css-syntax-3” in the subject, preferably like this: “[css-syntax-3] *…summary of comment…*”

Publication as a Candidate Recommendation does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This specification will not progress to Proposed Recommendation before 20 August 2014. See the section “CR exit criteria” for details.

For changes since the last draft, see the Changes section.

Table of Contents

1. Introduction
1.1. Module interactions

2. Description of CSS’s Syntax
2.1. Escaping
2.2. Error Handling

3. Tokenizing and Parsing CSS
3.1. Overview of the Parsing Model
3.2. The input byte stream
3.3. Preprocessing the input stream

4. Tokenization
4.1. Token Railroad Diagrams
4.2. Definitions
4.3. Tokenizer Algorithms
4.3.1. Consume a token
4.3.2. Consume a numeric token
4.3.3. Consume an ident-like token
4.3.4. Consume a string token
4.3.5. Consume a url token
4.3.6. Consume a unicode-range token
4.3.7. Consume an escaped code point
4.3.8. Check if two code points are a valid escape
4.3.9. Check if three code points would start an identifier
4.3.10. Check if three code points would start a number
4.3.11. Consume a name

4.3.12. Consume a number
4.3.13. Convert a string to a number
4.3.14. Consume the remnants of a bad url

5. Parsing
5.1. Parser Railroad Diagrams
5.2. Definitions
5.3. Parser Entry Points
5.3.1. Parse a stylesheet
5.3.2. Parse a list of rules
5.3.3. Parse a rule
5.3.4. Parse a declaration
5.3.5. Parse a list of declarations
5.3.6. Parse a component value
5.3.7. Parse a list of component values
5.4. Parser Algorithms
5.4.1. Consume a list of rules
5.4.2. Consume an at-rule
5.4.3. Consume a qualified rule
5.4.4. Consume a list of declarations
5.4.5. Consume a declaration
5.4.6. Consume a component value
5.4.7. Consume a simple block
5.4.8. Consume a function

6. The *An+B* microsyntax
6.1. Informal Syntax Description
6.2. The `<an+b>` type

7. Defining Grammars for Rules and Other Values
7.1. Defining Block Contents: the <declaration-list>, <rule-list>, and <stylesheet> productions

8. CSS stylesheets
8.1. Style rules
8.2. The @charset Rule

9. Serialization
9.1. Serializing *<an+b>*

10. Changes
10.1. Changes from the 5 November 2013 Last Call Working Draft
10.2. Changes from the 19 September 2013 Working Draft
10.3. Changes from CSS 2.1 and Selectors Level 3

Acknowledgments

Conformance
10.4. Document conventions
10.5. Conformance classes

10.6. Partial implementations
10.7. Experimental implementations
10.8. Non-experimental implementations
10.9. CR exit criteria

References
Normative References
Informative References

Index

Property index

§ 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 qualified rules, which are usually style rules that apply CSS properties to elements, 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. 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 identifiers, which have to start with a letter or a hyphen followed by a letter, and then can contain letters, numbers, hyphens, or underscores. 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 identifier 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 identifier 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.

§ 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 a ***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 the stream of bytes into a stream of code points, UAs must use the ***decode*** algorithm defined in [ENCODING], with the fallback encoding determined as follows.

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***:

1. If HTTP or equivalent protocol defines an encoding (e.g. via the charset parameter of the Content-Type header), ***get an encoding*** [ENCODING] for the specified value. If that does not return failure, use the return value as the fallback encoding.
2. Otherwise, check the 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 for the sequence of `XX` bytes, interpreted as `ASCII`.

   ▶ *What does that byte sequence mean?*

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

   ▶ *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 '@charset' rule, but it's actually much more restrictive. A number of things you can do in CSS that would produce a valid '@charset' rule, 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, use that as the fallback encoding.
4. Otherwise, use `utf-8` as the fallback encoding.

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: [CSS3CASCADE] defines the environment encoding for `@import`.

§ 3.3. Preprocessing the input stream

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

Before sending the input stream to the tokenizer, implementations must make the following code point substitutions:

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

§ 4. Tokenization

Implementations must act as if they used the following algorithms to tokenize CSS. To transform a stream of code points into a stream of tokens, repeatedly consume a token until an <EOF-token> is reached, collecting the returned tokens into a stream. 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 the 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>***, ***<include-match-token>***, ***<dash-match-token>***, ***<prefix-match-token>***, ***<suffix-match-token>***, ***<substring-match-token>***, ***<column-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 representation composed of one or more code points, and a numeric value. <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 ***start*** and an ***end***, a pair of integers.

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.

Note: As a technical note, the tokenizer defined here requires only three code points of look-ahead. The tokens it produces are designed to allow Selectors to be parsed with one token of look-ahead, and additional tokens may be added in the future to maintain this invariant.

§ 4.1. Token Railroad Diagrams

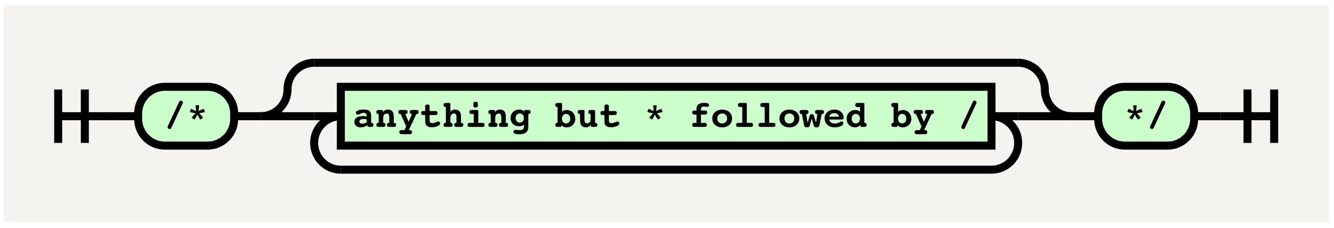
This section is non-normative.

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.

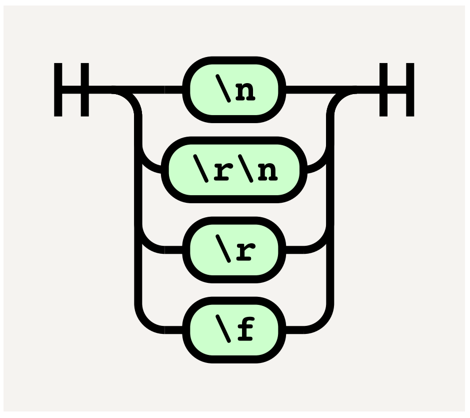
These diagrams are *informative* and *incomplete*; they describe the grammar of "correct" tokens, but do not describe error-handling at all. They are provided solely to make it easier to get an intuitive grasp of the syntax of each token.

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

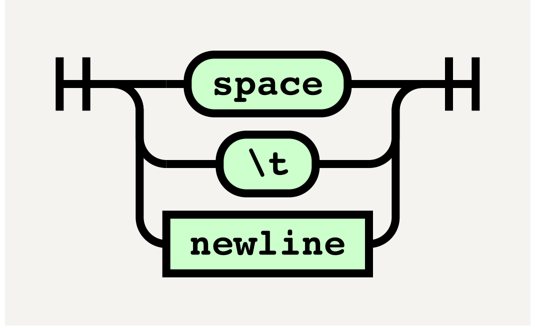
¶ **comment**



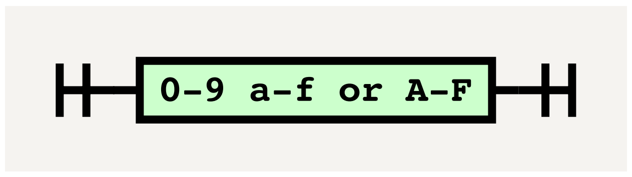
¶ **newline**



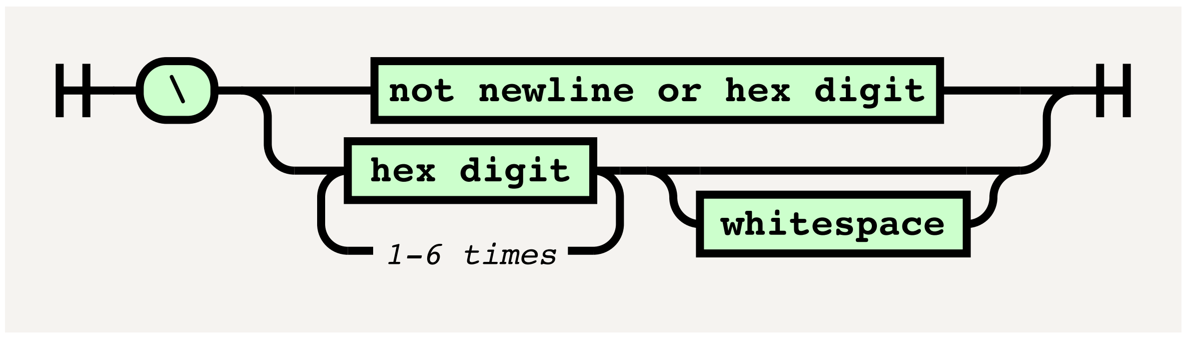
¶ **whitespace**



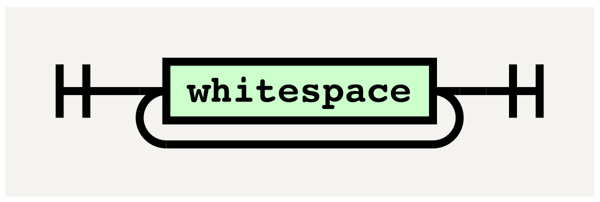
¶ hex digit



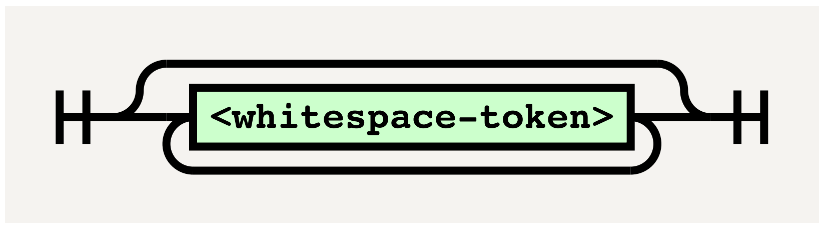
¶ escape



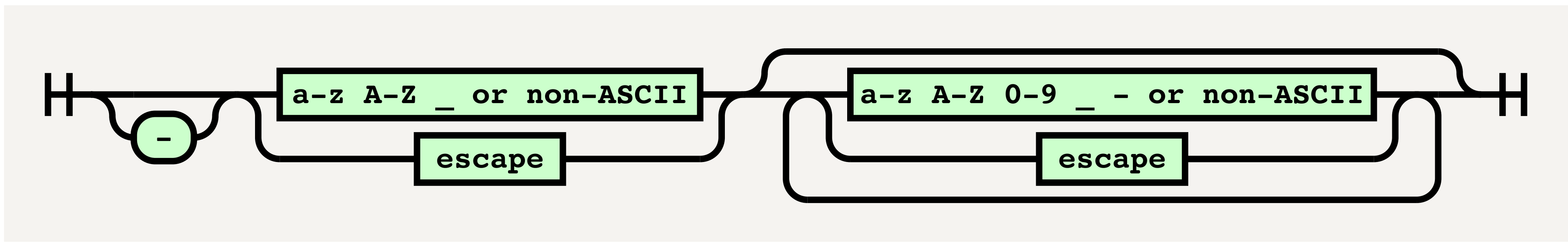
¶ <whitespace-token>



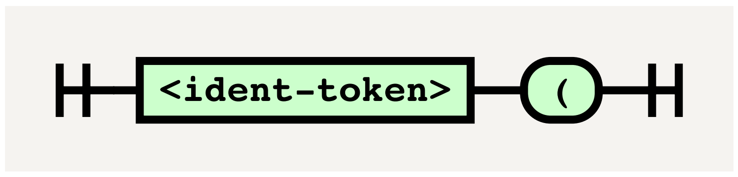
¶ ws*



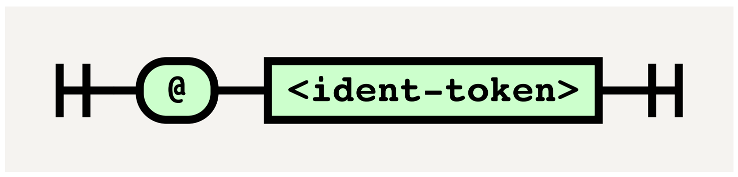
¶ <ident-token>



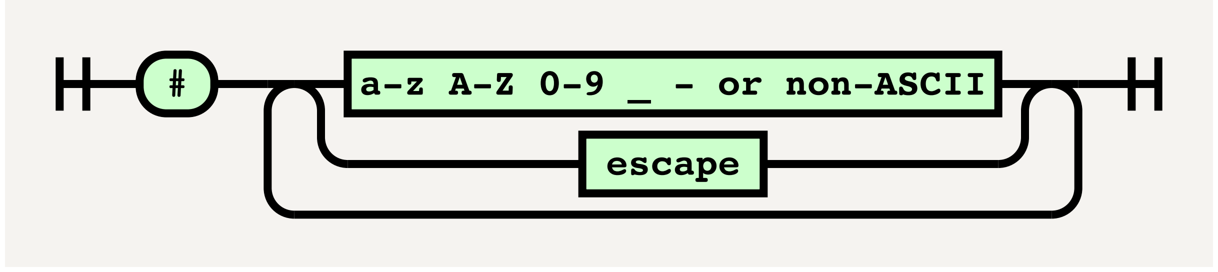
¶ <function-token>



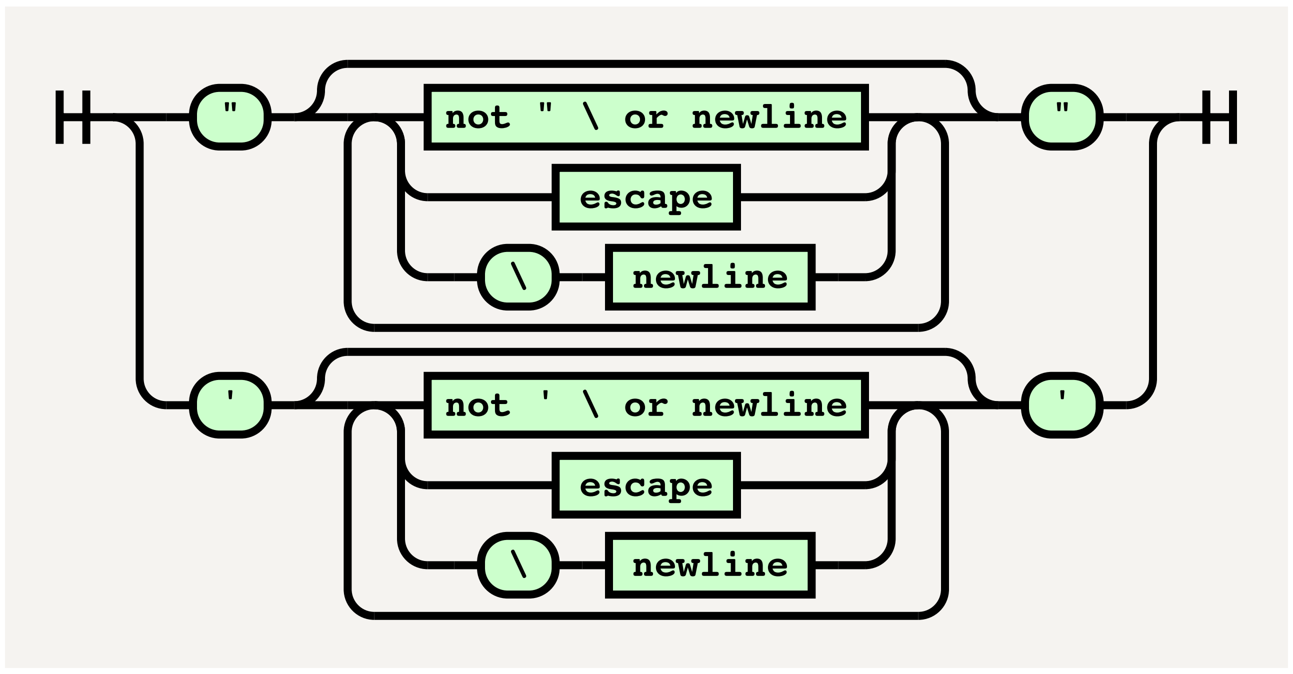
¶ <at-keyword-token>



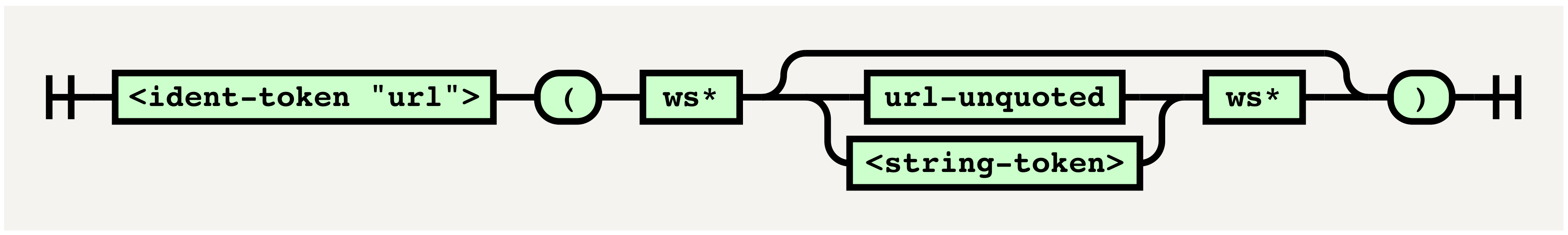
¶ <hash-token>



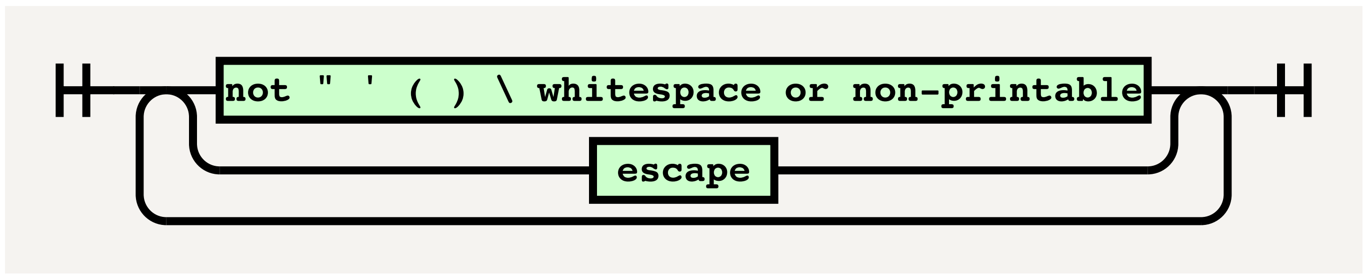
¶ **<string-token>**



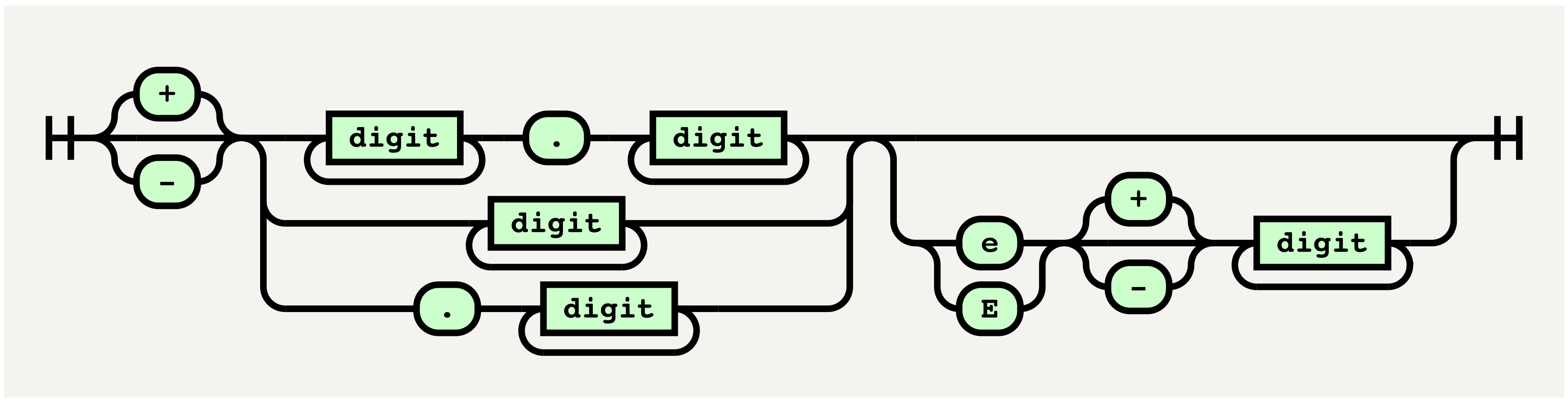
¶ **<url-token>**



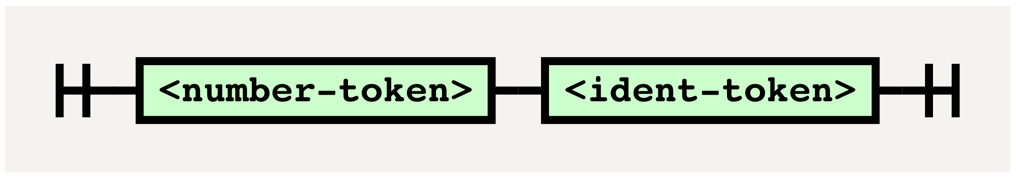
¶ **url-unquoted**



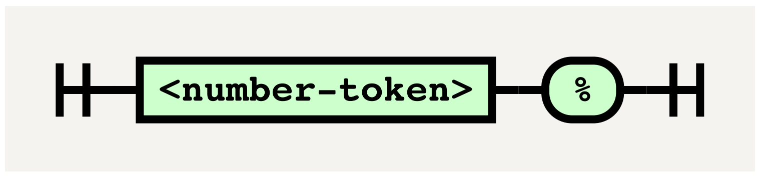
¶ **<number-token>**



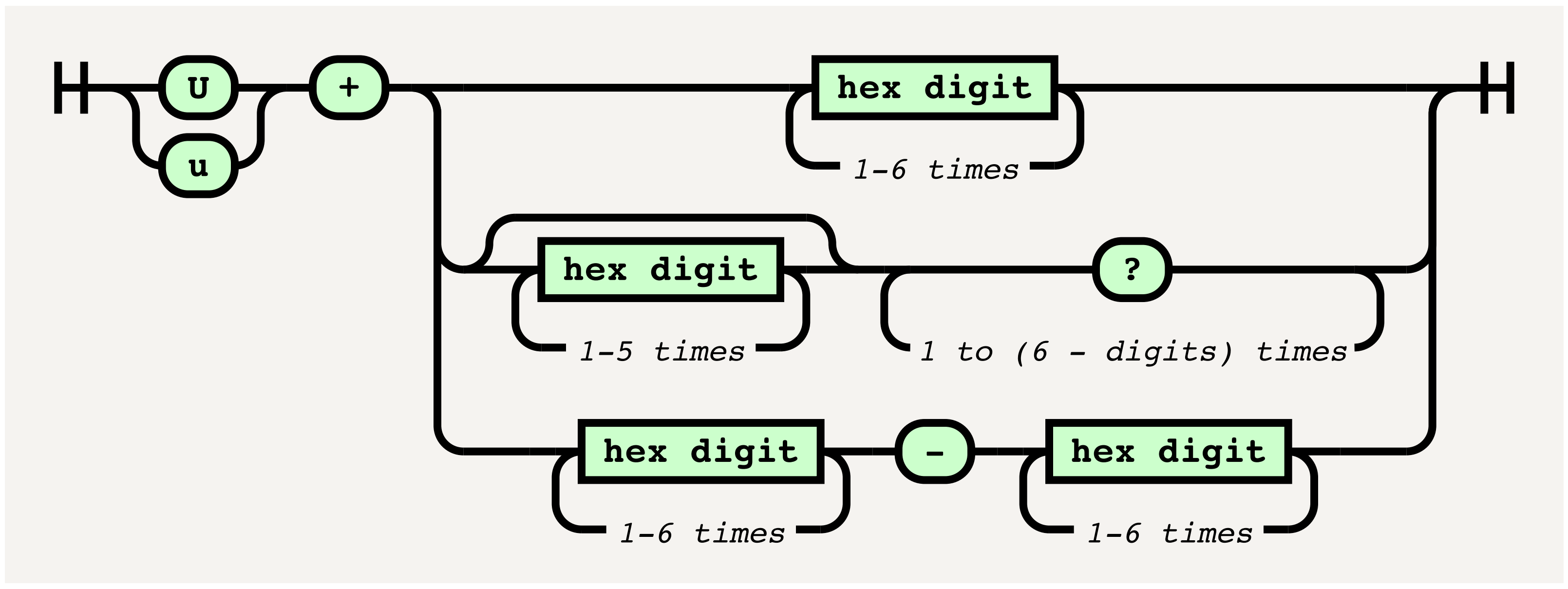
¶ **<dimension-token>**



¶ **<percentage-token>**

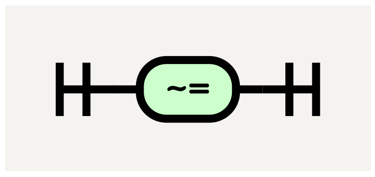


¶ **<unicode-range-token>**

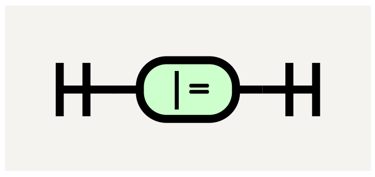


U
u
+
hex digit
1-6 times
hex digit
1-5 times
?
1 to (6 - digits) times
hex digit
1-6 times
-
hex digit
1-6 times

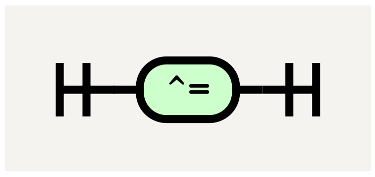
¶ **<include-match-token>**


~=

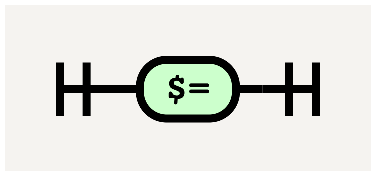
¶ **<dash-match-token>**


|=

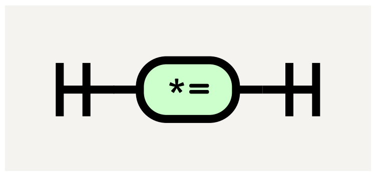
¶ **<prefix-match-token>**


^=

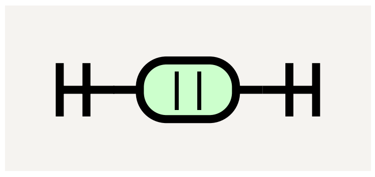
¶ **<suffix-match-token>**


$=

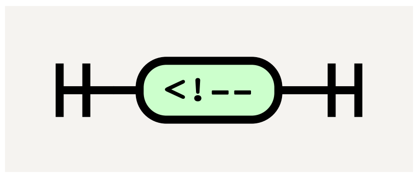
¶ **<substring-match-token>**


*=

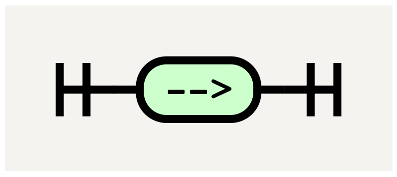
¶ **<column-token>**


||

¶ **<CDO-token>**


<!--

¶ **<CDC-token>**


-->

§ 4.2. Definitions

This section defines several terms used during the tokenization phase.

code point
: A Unicode code point. [UNICODE] Any value in the Unicode codespace; that is, the range of integers from 0 to (hexadecimal) 10FFFF.

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).

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

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

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

letter
: An uppercase letter or a lowercase letter.

non-ASCII code point
: A code point with a value equal to or greater than U+0080 <control>.

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

name code point
: A name-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, or U+000B LINE TABULATION, or

a code point between U+000E SHIFT OUT and U+001F INFORMATION SEPARATOR ONE, 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.

surrogate code point
: A code point between U+D800 and U+DFFF inclusive.

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

identifier
: A portion of the CSS source that has the same syntax as an <ident-token>. Also appears in <at-keyword-token>, <function-token>, <hash-token> with the "id" type flag, and the unit of <dimension-token>.

§ 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 will return a single token of any type.

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 with the ending code point U+0022 QUOTATION MARK (") and return it.

U+0023 NUMBER SIGN (#)
: If the next input code point is a name 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 identifier, set the <hash-token>'s type flag to "id".
3. Consume a name, 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+0024 DOLLAR SIGN ($)

If the next input code point is U+003D EQUALS SIGN (=), consume it and return a <suffix-match-token>.

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

U+0027 APOSTROPHE (')

Consume a string token with the ending code point U+0027 APOSTROPHE (') and return it.

U+0028 LEFT PARENTHESIS (()

Return a <(-token>.

U+0029 RIGHT PARENTHESIS ())

Return a <)-token>.

U+002A ASTERISK (*)

If the next input code point is U+003D EQUALS SIGN (=), consume it and return a <substring-match-token>.

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

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 input stream starts with an identifier, reconsume the current input code point, consume an ident-like 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, 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+002F SOLIDUS (/)

If the next input code point is U+002A ASTERISK (*), consume it 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. Then consume a 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 identifier, consume a name, 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+005E CIRCUMFLEX ACCENT (^)
If the next input code point is U+003D EQUALS SIGN (=), consume it and return a <prefix-match-token>.

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

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 SMALL LETTER U (u)
If the next 2 input code points are U+002B PLUS SIGN (+) followed by a hex digit or U+003F QUESTION MARK (?), consume the next input code point. Note: don't consume both of them. Consume a unicode-range token and return it.

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

name-start code point

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

U+007C VERTICAL LINE (|)

If the next input code point is U+003D EQUALS SIGN (=), consume it and return a <dash-match-token>.

Otherwise, if the next input code point is U+0073 VERTICAL LINE (|), consume it and return a <column-token>.

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

U+007E TILDE (~)

If the next input code point is U+003D EQUALS SIGN (=), consume it and return an <include-match-token>.

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

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 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.

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

1. Create a <dimension-token> with the same representation, value, and type flag as the returned number, and a unit set initially to the empty string.
2. Consume a name. 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 representation and value as the returned number, and return it.

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

§ 4.3.3. 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 a name.

If the returned string's value is an ASCII case-insensitive match for "url", and the next input code point is U+0028 LEFT PARENTHESIS ((), consume it. 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 the returned string and return it.

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

§ 4.3.4. 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 must be called with an *ending code point*, which denotes the code point that ends the string.

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
EOF

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, if 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.

§ 4.3.5. 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.

Execute the following steps in order:

1. Initially create a <url-token> with its value set to the empty string.
2. Consume as much whitespace as possible.
3. If the next input code point is EOF, return the <url-token>.
4. If the next input code point is a U+0022 QUOTATION MARK (") or U+0027 APOSTROPHE (‘), then:
    1. Consume a string token with the current input code point as the ending code point.
    2. If a <bad-string-token> was returned, consume the remnants of a bad url, create a <bad-url-token>, and return it.
    3. Set the <url-token>’s value to the returned <string-token>’s value.
    4. Consume as much whitespace as possible.
    5. If the next input code point is U+0029 RIGHT PARENTHESIS ()) or EOF, consume it and return the <url-token>; otherwise, consume the remnants of a bad url, create a <bad-url-token>, and return it.
5. Repeatedly consume the next input code point from the stream:

    U+0029 RIGHT PARENTHESIS ())
    EOF

    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>; 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.

§ 4.3.6. Consume a unicode-range token

This section describes how to ***consume a unicode-range token***. It returns a <unicode-range-token>.

Note: This algorithm assumes that the initial "u+" has been consumed, and the next code point verified to be a hex digit or a "?".

Execute the following steps in order:

1. 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.

   If any U+003F QUESTION MARK (?) code points were consumed, then:

   1. Interpret the consumed code points as a hexadecimal number, with the U+003F QUESTION MARK (?) code points replaced by U+0030 DIGIT ZERO (0) code points. This is the start of the range.
   2. Interpret the consumed code points as a hexadecimal number again, with the U+003F QUESTION MARK (?) code point replaced by U+0046 LATIN CAPITAL LETTER F (F) code points. This is the end of the range.
   3. Return a new <unicode-range-token> with the above start and end.

   Otherwise, interpret the digits as a hexadecimal number. This is the start of the range.

2. If the next 2 input code point 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 digits as a hexadecimal number. This is the end of the range.
3. Otherwise, the end of the range is the start of the range.
4. Return the <unicode-range-token> with the above start and end.

§ 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 not be a newline. 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 code point, or is greater than the maximum allowed code point, return U+FFFD REPLACEMENT CHARACTER (�). Otherwise, return the code point with that value.

EOF code point

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+005D 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 identifier

This section describes how to ***check if three code points would start an identifier***. 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 a name-start code point or the second and third code points are a valid escape, return true. Otherwise, return false.

name-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.

§ 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. Consume a name

This section describes how to ***consume a name*** 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 identifier before calling this algorithm.

Let *result* initially be an empty string.

Repeatedly consume the next input code point from the stream:

name 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

Return *result*.

§ 4.3.12. Consume a number

This section describes how to ***consume a number*** from a stream of code points. It returns a 3-tuple of a string representation, a numeric value, and a type flag which is either "integer" or "number".

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. Initially set *repr* to the empty string and *type* to "integer".
2. If the next input code point is U+002B PLUS SIGN (+) or U+002D HYPHEN-MINUS (-), consume it and append it to *repr*.
3. While the next input code point is a digit, consume it and append it to *repr*.
4. If the next 2 input code points are U+002E FULL STOP (.) followed by a digit, then:
    1. Consume them.
    2. Append them to *repr*.
    3. Set *type* to "number".
    4. While the next input code point is a digit, consume it and append it to *repr*.
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 them.
    2. Append them to *repr*.
    3. Set *type* to "number".
    4. While the next input code point is a digit, consume it and append it to *repr*.
6. Convert *repr* to a number, and set the *value* to the returned value.
7. Return a 3-tuple of *repr*, *value*, and *type*.

§ 4.3.13. Convert a string to a number

This section describes how to ***convert a string to a number***. It returns a number.

Note: This algorithm does not do any verification to ensure that the string contains only a number. Ensure that the string contains only a valid CSS number before calling this algorithm.

Divide the string into seven components, in order from left to right:

1. A **sign**: a single U+002B PLUS SIGN (+) or U+002D HYPHEN-MINUS (-), or the empty string. Let *s* be the number -1 if the sign is U+002D HYPHEN-MINUS (-); otherwise, let *s* be the number 1.
2. An **integer part**: zero or more digits. If there is at least one digit, let *i* be the number formed by interpreting the digits as a base-10 integer; otherwise, let *i* be the number 0.
3. A **decimal point**: a single U+002E FULL STOP (.), or the empty string.
4. A **fractional part**: zero or more digits. If there is at least one digit, let *f* be the number formed by interpreting the digits as a base-10 integer and *d* be the number of digits; otherwise, let *f* and *d*

be the number 0.

5. An **exponent indicator**: a single U+0045 LATIN CAPITAL LETTER E (E) or U+0065 LATIN SMALL LETTER E (e), or the empty string.
6. An **exponent sign**: a single U+002B PLUS SIGN (+) or U+002D HYPHEN-MINUS (-), or the empty string. Let *t* be the number -1 if the sign is U+002D HYPHEN-MINUS (-); otherwise, let *t* be the number 1.
7. An **exponent**: zero or more digits. If there is at least one digit, let *e* be the number formed by interpreting the digits as a base-10 integer; otherwise, let *e* be the number 0.

Return the number $s \cdot (i + f \cdot 10^{-d}) \cdot 10^{te}$.

§ 4.3.14. 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 input to the parsing stage is a stream or list of tokens from the tokenization stage. The output depends on how the parser is invoked, as defined by the entry points listed later in this section. The parser output can consist of at-rules, qualified rules, and/or declarations.

The parser’s output is constructed according to the fundamental syntax of CSS, without regards for the validity of any specific item. Implementations may check the validity of items as they are returned by the various parser algorithms and treat the algorithm as returning nothing if the item was invalid according to the implementation’s own grammar knowledge, or may construct a full tree as specified and "clean up" afterwards by removing any invalid items.

The items that can appear in the tree are:

at-rule

An at-rule has a name, a prelude consisting of a list of component values, and an optional block consisting of a simple {} block.

> Note: This specification places no limits on what an at-rule's block may contain. Individual at-rules must define whether they accept a block, and if so, how to parse it (preferably using one of the parser algorithms or entry points defined in this specification).

qualified rule

A qualified rule has a prelude consisting of a list of component values, and a block consisting of a simple {} block.

> Note: Most qualified rules will be style rules, where the prelude is a selector [SELECT] and the block a list of declarations.

declaration

A declaration has a name, a value consisting of a list of component values, and an *important* flag which is initially unset.

Declarations are further categorized as "properties" or "descriptors", with the former typically appearing in qualified rules and the latter appearing 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-grainted 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

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

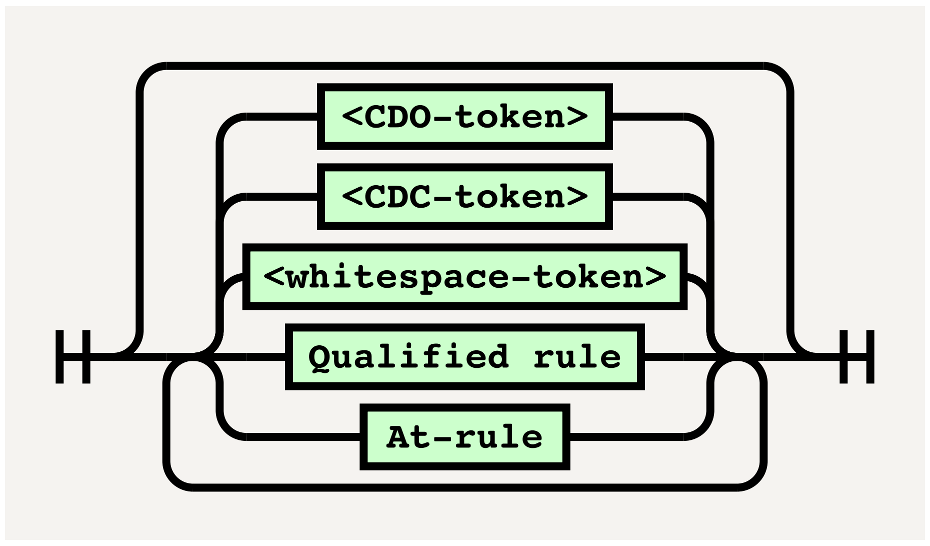
§ 5.1. Parser Railroad Diagrams

This section is non-normative.

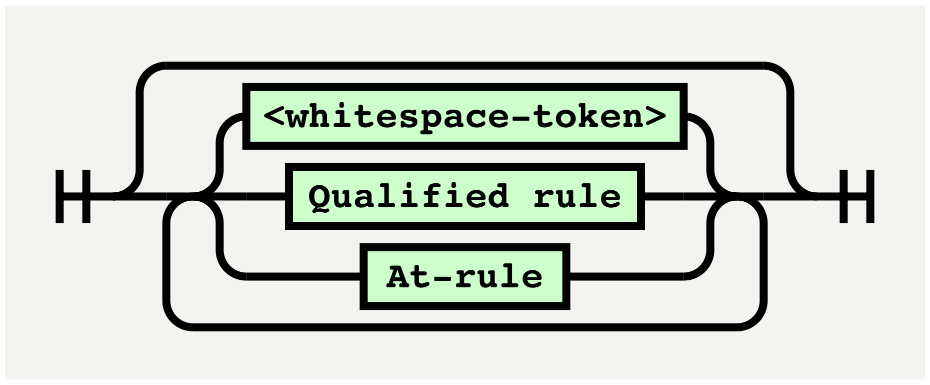
This section presents an informative view of the parser, in the form of railroad diagrams. Railroad diagrams are more compact than a state-machine, but often easier to read than a regular expression.

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.

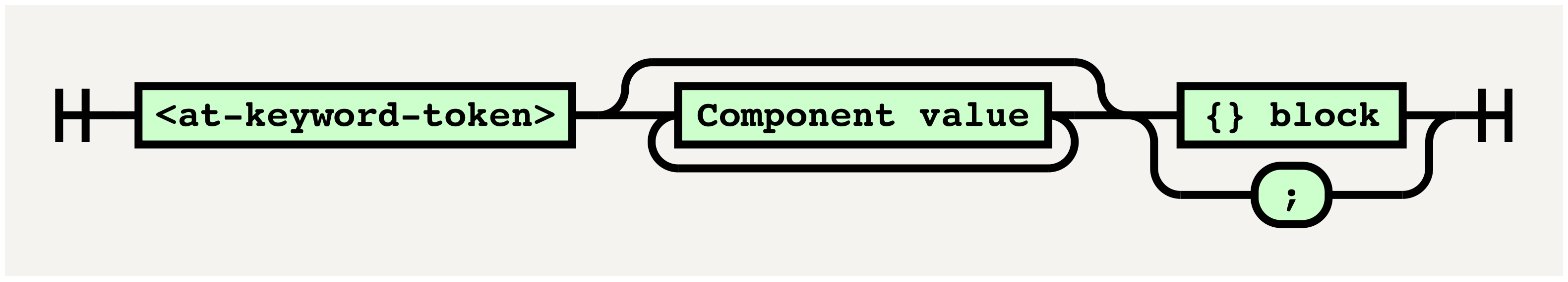
¶ **Stylesheet**



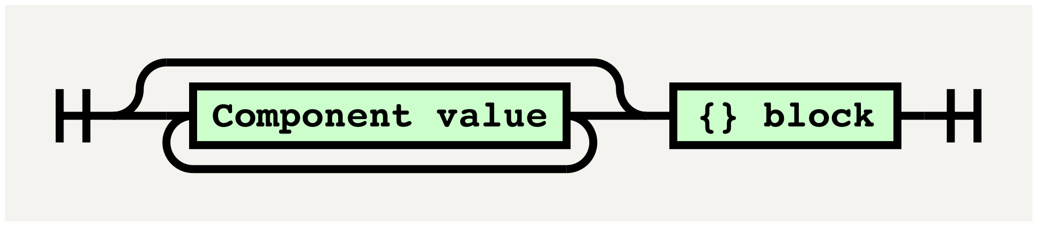
¶ **Rule list**



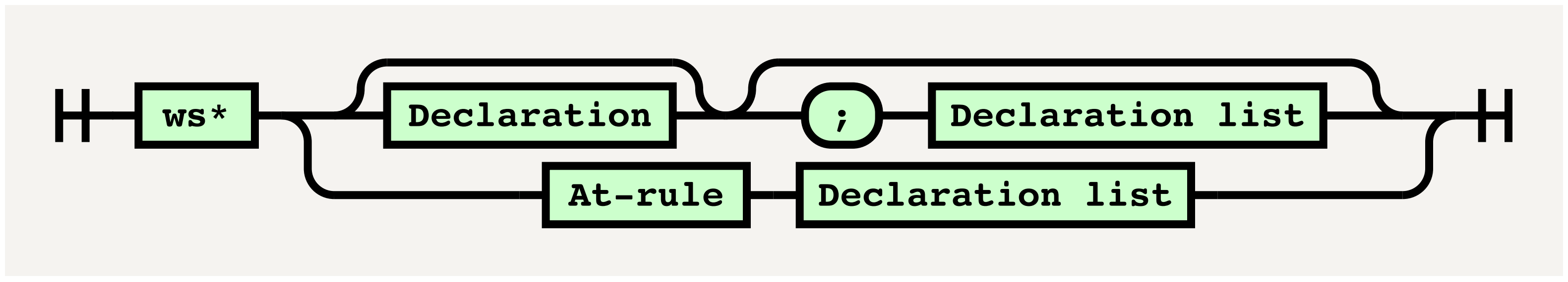
¶ **At-rule**



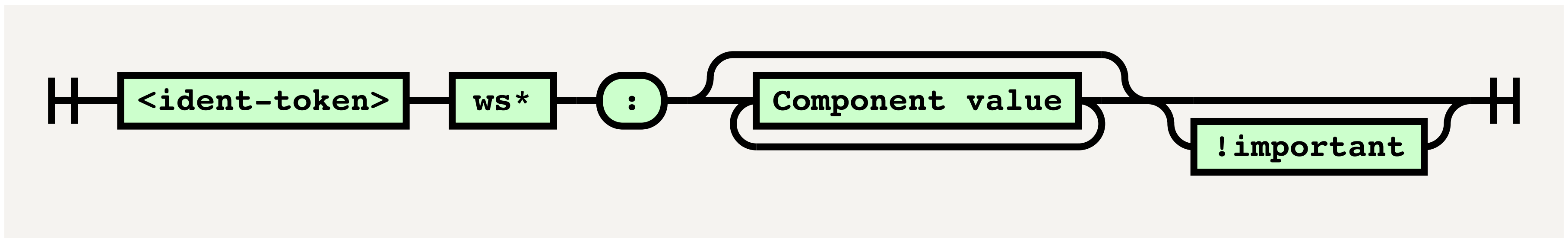
¶ **Qualified rule**



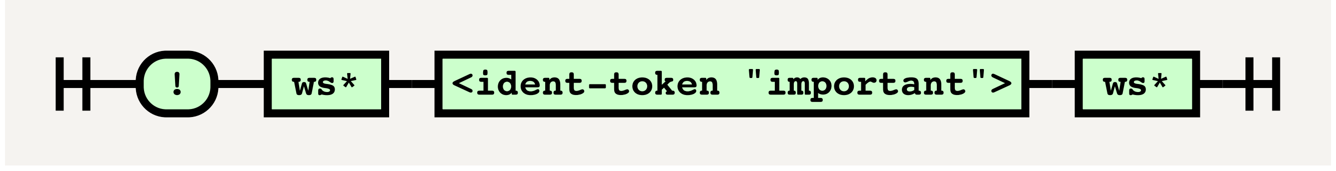
¶ **Declaration list**



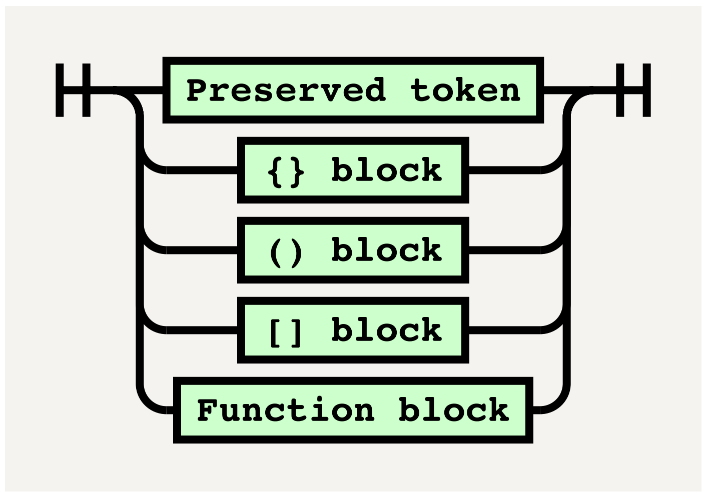
¶ **Declaration**



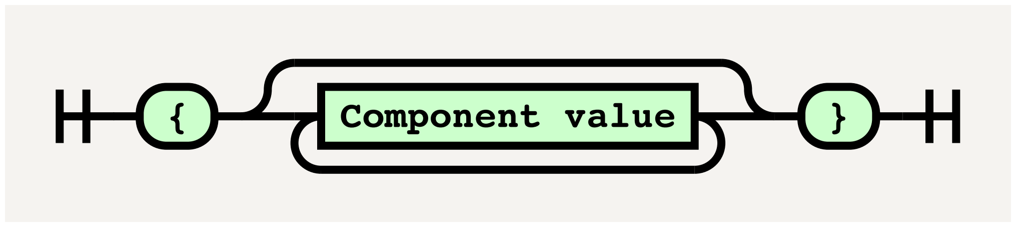
¶ **!important**



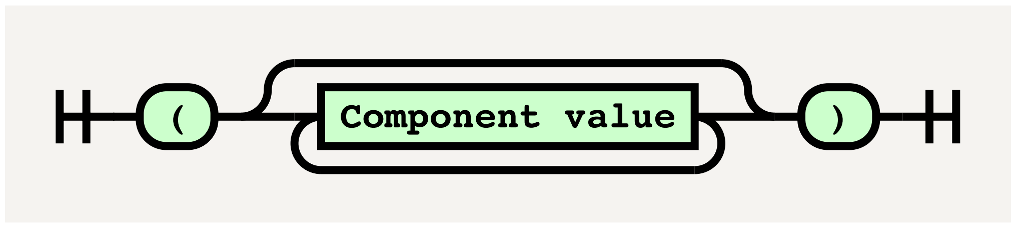
¶ **Component value**



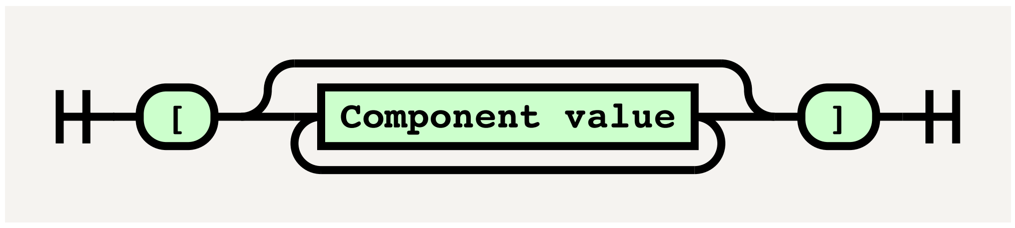
¶ **{} block**



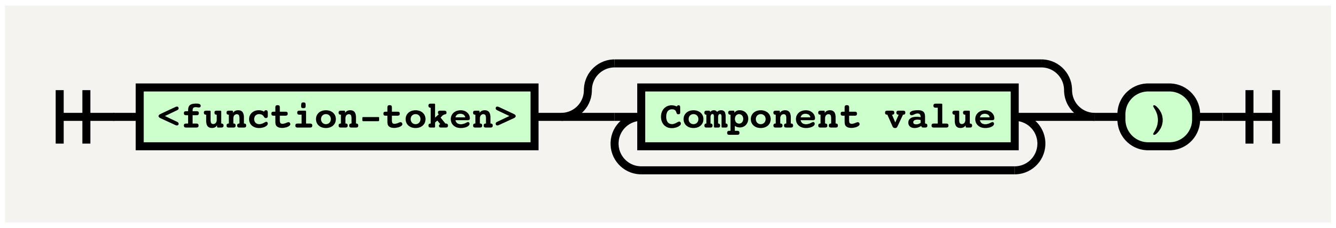
¶ **() block**



¶ **[] block**



¶ **Function block**



§ 5.2. Definitions

current input token

The token or component value currently being operated on, from the list of tokens produced by the tokenizer.

next input token

The token or component value following the current input token in the list of tokens produced by the tokenizer. If there isn't a token following the current input token, the next input token is an <EOF-token>.

<EOF-token>

A conceptual token representing the end of the list of tokens. Whenever the list of tokens is empty, the next input token is always an <EOF-token>.

consume the next input token

Let the current input token be the current next input token, adjusting the next input token accordingly.

reconsume the current input token

The next time an algorithm instructs you to consume the next input token, instead do nothing (retain the current input token unchanged).

ASCII case-insensitive

When two strings are to be matched ASCII case-insensitively, temporarily convert both of them to ASCII lower-case form by adding 32 (0x20) to the value of each code point between U+0041 LATIN CAPITAL LETTER A (A) and U+005A LATIN CAPITAL LETTER Z (Z), inclusive, and check if this results in identical sequences of code point.

§ 5.3. Parser Entry Points

The algorithms defined in this section produce high-level CSS objects from lower-level objects. They assume that they are invoked on a token stream, but they may also be invoked on a string; if so, first perform input preprocessing to produce a code point stream, then perform tokenization to produce a token stream.

"Parse a stylesheet" can also be invoked on a byte stream, in which case The input byte stream defines how to decode it into Unicode.

Note: This specification does not define how a byte stream is decoded for other entry points.

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 list of rules" is intended for the content of at-rules such as '@media'. It differs from "Parse a stylesheet" in the handling of <CDO-token> and <CDC-token>.
- "Parse a rule" is intended for use by the `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 list of declarations" is for the contents of a `style` attribute, which parses text into the contents of a single style rule.
- "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.

All of the algorithms defined in this spec may be called with either a list of tokens or of component values. Either way produces an identical result.

§ 5.3.1. Parse a stylesheet

To ***parse a stylesheet*** from a stream of tokens:

1. Create a new stylesheet.
2. Consume a list of rules from the stream of tokens, with the *top-level flag* set.
3. Assign the returned value to the stylesheet’s value.
4. Return the stylesheet.

§ 5.3.2. Parse a list of rules

To ***parse a list of rules*** from a stream of tokens:

1. Consume a list of rules from the stream of tokens, with the *top-level flag* unset.
2. Return the returned list.

§ 5.3.3. Parse a rule

To ***parse a rule*** from a stream of tokens:

1. Consume the next input token.
2. While the current input token is a <whitespace-token>, consume the next input token.
3. If the current input token is an <EOF-token>, return a syntax error.

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

   Otherwise, consume a qualified rule and let *rule* be the return value. If nothing was returned, return a syntax error.
4. While the current input token is a <whitespace-token>, consume the next input token.
5. If the current input token is an <EOF-token>, return *rule*. Otherwise, return a syntax error.

§ 5.3.4. Parse a declaration

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

To ***parse a declaration***:

1. Consume the next input token.
2. While the current input token is a <whitespace-token>, consume the next input token.
3. If the current input token is not an <ident-token>, return a syntax error.
4. Consume a declaration. If anything was returned, return it. Otherwise, return a syntax error.

§ 5.3.5. Parse a list of declarations

Note: Despite the name, this actually parses a mixed list of declarations and at-rules, as CSS 2.1 does for ‘@page’. Unexpected at-rules (which could be all of them, in a given context) are invalid and should be ignored by the consumer.

To ***parse a list of declarations***:

1. Consume a list of declarations.
2. Return the returned list.

§ 5.3.6. Parse a component value

To ***parse a component value***:

1. Consume the next input token.
2. While the current input token is a <whitespace-token>, consume the next input token.
3. If the current input token is an <EOF-token>, return a syntax error.
4. Reconsume the current input token. Consume a component value and let *value* be the return value. If nothing is returned, return a syntax error.
5. While the current input token is a <whitespace-token>, consume the next input token.
6. If the current input token is an <EOF-token>, return *value*. Otherwise, return a syntax error.

§ 5.3.7. Parse a list of component values

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

1. Repeatedly consume a component value until an <EOF-token> is returned, appending the returned values (except the final <EOF-token>) into a list. Return the list.

§ 5.4. Parser Algorithms

The following algorithms comprise the parser. They are called by the parser entry points above.

These algorithms may be called with a list of either tokens or of component values. (The difference being that some tokens are replaced by functions and simple blocks in a list of component values.) Similar to how the input stream returned EOF code points to represent when it was empty during the tokenization stage, the lists in this stage must return an <EOF-token> when the next token is requested but they are empty.

An algorithm may be invoked with a specific list, in which case it consumes only that list (and when that list is exhausted, it begins returning <EOF-token>s). Otherwise, it is implicitly invoked with the same list as the invoking algorithm.

§ 5.4.1. Consume a list of rules

To ***consume a list of rules***:

Create an initially empty list of rules.

Repeatedly consume the next input token:

<whitespace-token>
: Do nothing.

<EOF-token>
: Return the list of rules.

<CDO-token>
<CDC-token>
: If the ***top-level flag*** is set, do nothing.

  Otherwise, reconsume the current input token. Consume a qualified rule. If anything is returned, append it to the list of rules.

<at-keyword-token>
: Reconsume the current input token. Consume an at-rule. If anything is returned, append it to the list of rules.

anything else
: Reconsume the current input token. Consume a qualified rule. If anything is returned, append it to the list of rules.

§ 5.4.2. Consume an at-rule

To ***consume an at-rule***:

Create a new at-rule with its name set to the value of the current input token, its prelude initially set to an empty list, and its value initially set to nothing.

Repeatedly consume the next input token:

<semicolon-token>
<EOF-token>
: Return the at-rule.

<{-token>
: Consume a simple block and assign it to the at-rule’s block. Return the at-rule.

simple block with an associated token of <{-token>
: Assign the block to the at-rule’s block. Return the at-rule.

anything else
: Reconsume the current input token. Consume a component value. Append the returned value to the at-rule’s prelude.

§ 5.4.3. Consume a qualified rule

To ***consume a qualified rule***:

Create a new qualified rule with its prelude initially set to an empty list, and its value initially set to nothing.

Repeatedly consume the next input token:

<EOF-token>
: This is a parse error. Return nothing.

<{-token>
: Consume a simple block and assign it to the qualified rule’s block. Return the qualified rule.

simple block with an associated token of <{-token>
: Assign the block to the qualified rule’s block. Return the qualified rule.

anything else
: Reconsume the current input token. Consume a component value. Append the returned value to the qualified rule’s prelude.

§ 5.4.4. Consume a list of declarations

To ***consume a list of declarations***:

Create an initially empty list of declarations.

Repeatedly consume the next input token:

<whitespace-token>
<semicolon-token>
: Do nothing.

<EOF-token>
: Return the list of declarations.

<at-keyword-token>
: Consume an at-rule. Append the returned rule to the list of declarations.

<ident-token>

Initialize a temporary list initially filled with the current input token. Consume the next input token. While the current input token is anything other than a <semicolon-token> or <EOF-token>, append it to the temporary list and consume the next input token. Consume a declaration from the temporary list. If anything was returned, append it to the list of declarations.

anything else

This is a parse error. Repeatedly consume a component value until it is a <semicolon-token> or <EOF-token>.

§ 5.4.5. Consume a declaration

To ***consume a declaration***:

Create a new declaration with its name set to the value of the current input token and its value initially set to the empty list.

1. Consume the next input token.
2. While the current input token is a <whitespace-token>, consume the next input token.
3. If the current input token is anything other than a <colon-token>, this is a parse error. Return nothing.

   Otherwise, consume the next input token.
4. While the current input token is anything other than an <EOF-token>, append it to the declaration's value and consume the next input token.
5. If the last two non-<whitespace-token>s in the declaration'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 the declaration's value and set the declaration's *important* flag to true.
6. Return the declaration.

§ 5.4.6. Consume a component value

To ***consume a component value***:

Consume the next input token.

If the current input token is a <{-token>, <[-token>, or <(-token>, consume a simple block and return it.

Otherwise, if the current input token is a <function-token>, consume a function and return it.

Otherwise, return the current input token.

§ 5.4.7. Consume a simple block

To ***consume a simple block***:

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

Create a simple block with its associated token set to the current input token and with a value with is initially an empty list.

Repeatedly consume the next input token and process it as follows:

<EOF-token>
ending token

Return the block.

anything else

Reconsume the current input token. Consume a component value and append it to the value of the block.

§ 5.4.8. Consume a function

To ***consume a function***:

Create a function with a name equal to the value of the current input token, and with a value which is initially an empty list.

Repeatedly consume the next input token and process it as follows:

<EOF-token>
<)-token>

Return the function.

anything else

Reconsume the current input token. Consume a component value and append the returned value to the function’s value.

§ 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:

```
-n+6   /* represents the first 6 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 */
```

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

EXAMPLE 9

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 whose representation starts with "+" or "-"
- ***<signless-integer>*** is a <number-token> with its type flag set to "integer", and whose representation start with a digit

†: 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.

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

The Values spec defines how to specify a grammar for properties. This section does the same, but for rules.

Just like in property grammars, the notation `<foo>` refers to the "foo" grammar term, assumed to be defined elsewhere. Substituting the `<foo>` for its definition results in a semantically identical grammar.

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 an ASCII case-insensitive match for the value defined in the grammar.

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 10

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 <declaration-list>, <rule-list>, and <stylesheet> productions

The CSS parser is agnostic as to the contents of blocks, such as those that come at the end of some at-rules. Defining the generic grammar of the blocks in terms of tokens is non-trivial, but there are dedicated and unambiguous algorithms defined for parsing this.

The ***<declaration-list>*** production represents a list of declarations. It may only be used in grammars as the sole value in a block, and represents that the contents of the block must be parsed using the consume a list of declarations algorithm.

Similarly, the ***<rule-list>*** production represents a list of rules, and may only be used in grammars as the sole value in a block. It represents that the contents of the block must be parsed using the consume a list of rules algorithm.

Finally, the ***<stylesheet>*** production represents a list of rules. It is identical to <rule-list>, except that blocks using it default to accepting all rules that aren’t otherwise limited to a particular context.

EXAMPLE 11

For example, the ‘@font-face’ rule is defined to have an empty prelude, and to contain a list of declarations. This is expressed with the following grammar:

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

This is a complete and sufficient definition of the rule’s grammar.

For another example, ‘@keyframes’ rules are more complex, interpreting their prelude as a name and containing keyframes rules in their block Their grammar is:

```
@keyframes <keyframes-name> { <rule-list> }
```

For rules that use <declaration-list>, the spec for the rule must define which properties, descriptors, and/or at-rules are valid inside the rule; this may be as simple as saying "The @foo rule accepts the properties/descriptors defined in this specification/section.", and extension specs may simply say "The @foo rule additionally accepts the following properties/descriptors.". Any declarations or at-rules found inside the block that are not defined as valid must be removed from the rule's value.

Within a <declaration-list>, `!important` is automatically invalid on any descriptors. If the rule accepts properties, the spec for the rule must define whether the properties interact with the cascade, and with what specificity. If they don't interact with the cascade, properties containing `!important` are automatically invalid; otherwise using `!important` is valid and has its usual effect on the cascade origin of the property.

> EXAMPLE 12
> For example, the grammar for '@font-face' in the previous example must, in addition to what is written there, define that the allowed declarations are the descriptors defined in the Fonts spec.

For rules that use <rule-list>, the spec for the rule must define what types of rules are valid inside the rule, same as <declaration-list>, and unrecognized rules must similarly be removed from the rule's value.

> EXAMPLE 13
> For example, the grammar for '@keyframes' in the previous example must, in addition to what is written there, define that the only allowed rules are <keyframe-rule>s, which are defined as:
>
> ```
> <keyframe-rule> = <keyframe-selector> { <declaration-list> }
> ```
>
> Keyframe rules, then, must further define that they accept as declarations all animatable CSS properties, plus the 'animation-timing-function' property, but that they do not interact with the cascade.

For rules that use <stylesheet>, all rules are allowed by default, but the spec for the rule may define what types of rules are *invalid* inside the rule.

> EXAMPLE 14
> For example, the '@media' rule accepts anything that can be placed in a stylesheet, except more '@media' rules. As such, its grammar is:
>
> ```
> @media <media-query-list> { <stylesheet> }
> ```
>
> It additionally defines a restriction that the <stylesheet> can not contain '@media' rules, which causes them to be dropped from the outer rule's value if they appear.

§ 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 [SELECT] with a list of property declarations. They are also called rule sets in [CSS21]. CSS Cascading and Inheritance [CSS3CASCADE] 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 results in an invalid selector list, the entire style rule is invalid.

The content of the qualified rule's block is parsed as a list of declarations. Unless defined otherwise by another specification or a future level of this specification, at-rules in that list are invalid and must be ignored. Declaration 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 15

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

§ 8.2. The '@charset' Rule

The @charset rule is an artifact of the algorithm used to determine the fallback encoding for the stylesheet. That algorithm looks for a specific byte sequence as the very first few bytes in the file, which has the syntactic form of an @-rule. Those bytes are not discarded from the input, whether or not they influence the encoding actually used to process the stylesheet.

Therefore, the stylesheet parser recognizes an @-rule with the general syntax

```
@charset = @charset <string> ;
```

and, for backward compatibility, includes it in the object model for the stylesheet. Modifying, adding, or removing an @charset rule via the object model has no effect (in particular it does **not** cause the stylesheet to be rescanned in a different encoding). The @charset rule is invalid if it is not the very first, top-level rule in the stylesheet, but it is parsed according to the normal syntax for @-rules, which are less restrictive than the algorithm that determines the fallback encoding. Therefore, an @charset rule may appear in the object model even if it was ignored by that algorithm. (For instance, if it was written with extra whitespace or with single rather than double quotes.)

§ 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 an 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 either of the following two tables, 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, the preserved comment 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**	**unicode range**	**CDC**	**(**	**?**
ident	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
at-keyword	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
hash	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
dimension	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		
#	✗	✗	✗	✗	✗	✗	✗	✗	✗			
-	✗	✗	✗	✗		✗	✗	✗	✗			
number	✗	✗	✗	✗		✗	✗	✗	✗			
@	✗	✗	✗	✗	✗				✗			
unicode range	✗	✗				✗	✗	✗				✗
.						✗	✗	✗				

+	✗	✗	✗

	=	|	*
$	✗		
*	✗		
^	✗		
~	✗		
|	✗	✗	
/			✗

§ 9.1. Serializing *<an+b>*

To serialize an *<an+b>* value, let *s* initially be the empty string:

***A* and *B* are both zero**
: Append "0" to *s*.

***A* is zero, *B* is non-zero**
: Serialize *B* and append it to *s*.

***A* is non-zero, *B* is zero**
: Serialize *A* and append it to *s*. Append "n" to *s*.

***A* and *B* are both non-zero**
: Serialize *A* and append it to *s*. Append "n" to *s*. If *B* is positive, append "+" to *s* Serialize *B* and append it to *s*.

Return *s*.

§ 10. Changes

This section is non-normative.

§ 10.1. 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";
```

§ 10.2. 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.

§ 10.3. 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 registery 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 code point was changed to be consistent with every definition of ASCII. This affects code points U+0080 to U+009F, which are now name code points rather than <delim-token>s, like the rest of non-ASCII code points.
- 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> is now more restrictive, and doesn't include non-sensical patterns that the 2.1 definition allowed.
- 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.
- <prefix-match-token>, <suffix-match-token>, and <substring-match-token> have been imported from Selectors 3.

- 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>, <number-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.
- <column-token> has been added, to keep Selectors parsing in single-token lookahead.
- Hexadecimal escape for surrogate code points now emit a replacement character rather than the surrogate. This allows implementations to safely use UTF-16 internally.

Parsing changes:

- Any list of declaration 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 and lets the next declaration being at a {} block without a <semicolon-token>.
- The handling of some miscellanous "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, Henri Sivonen, 呂康豪 (Kang-Hao Lu), Marc O’Morain, Simon Pieter, and Zack Weinberg.

§ Conformance

§ 10.4. 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 16

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.

§ 10.5. 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.

§ 10.6. 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.

§ 10.7. Experimental implementations

To avoid clashes with future CSS features, the CSS2.1 specification reserves a prefixed syntax for proprietary and experimental extensions to CSS.

Prior to a specification reaching the Candidate Recommendation stage in the W3C process, all implementations of a CSS feature are considered experimental. The CSS Working Group recommends that implementations use a vendor-prefixed syntax for such features, including those in W3C Working Drafts. This avoids incompatibilities with future changes in the draft.

§ 10.8. 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.

§ 10.9. CR exit criteria

For this specification to be advanced to Proposed Recommendation, there must be at least two independent, interoperable implementations of each feature. Each feature may be implemented by a different set of products, there is no requirement that all features be implemented by a single product. For the purposes of this criterion, we define the following terms:

independent
: each implementation must be developed by a different party and cannot share, reuse, or derive from code used by another qualifying implementation. Sections of code that have no bearing on the implementation of this specification are exempt from this requirement.

interoperable
: passing the respective test case(s) in the official CSS test suite, or, if the implementation is not a Web browser, an equivalent test. Every relevant test in the test suite should have an equivalent test created if such a user agent (UA) is to be used to claim interoperability. In addition if such a UA is to be used to claim interoperability, then there must one or more additional UAs which can also pass those equivalent tests in the same way for the purpose of interoperability. The equivalent tests must be made publicly available for the purposes of peer review.

implementation
: a user agent which: **(1)** implements the specification. **(2)** is available to the general public. The implementation may be a shipping product or other publicly available version (i.e., beta version, preview release, or "nightly build"). Non-shipping product releases must have implemented the feature(s) for a period of at least one month in order to demonstrate stability. **(3)** is not experimental (i.e., a version specifically designed to pass the test suite and is not intended for normal usage going forward).

The specification will remain Candidate Recommendation for at least six months.

§ References

§ Normative References

¶ **[CSS21]**
Bert Bos; et al. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. 7 June 2011. W3C Recommendation. URL: http://www.w3.org/TR/2011/REC-CSS2-20110607

¶ **[CSS3CASCADE]**
Håkon Wium Lie; Elika J. Etemad; Tab Atkins Jr.. CSS Cascading and Inheritance Level 3. 3 October 2013. W3C Candidate Recommendation. (Work in progress.) URL: http://www.w3.org/TR/2013/CR-css-cascade-3-20131003/

¶ **[ENCODING]**
Anne van Kesteren; Joshua Bell; Addison Phillips. Encoding. W3C 28 January 2014. First Public Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2014/WD-encoding-20140128/

¶ **[RFC2119]**
S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. URL: http://www.ietf.org/rfc/rfc2119.txt

¶ **[SELECT]**
Tantek Çelik; et al. Selectors Level 3. 29 September 2011. W3C Recommendation. URL: http://www.w3.org/TR/2011/REC-css3-selectors-20110929/

¶ **[UNICODE]**

The Unicode Consortium. The Unicode Standard. 2013. Defined by: The Unicode Standard, Version 6.3.0, (Mountain View, CA: The Unicode Consortium, 2013. ISBN 978-1-936213-08-5), as updated from time to time by the publication of new versions URL: http://www.unicode.org/standard/versions/enumeratedversions.html

§ Informative References

¶ **[CSS-VARIABLES]**
Tab Atkins Jr.; Luke Macpherson; Daniel Glazman. CSS Custom Properties for Cascading Variables Module Level 1. 20 June 2013. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2013/WD-css-variables-1-20130620/

¶ **[CSS3-ANIMATIONS]**
Dean Jackson; et al. CSS Animations. 19 February 2013. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2013/WD-css3-animations-20130219/

¶ **[CSS3-CONDITIONAL]**
L. David Baron. CSS Conditional Rules Module Level 3. 4 April 2013. W3C Candidate Recommendation. (Work in progress.) URL: http://www.w3.org/TR/2013/CR-css3-conditional-20130404/

¶ **[CSSOM]**
Anne van Kesteren. CSSOM. 12 July 2011. W3C Working Draft. (Work in progress.) URL: http://www.w3.org/TR/2011/WD-cssom-20110712/

¶ **[HTML]**
Robin Berjon, et al. HTML5. 4 February 2014. W3C Candidate Recommendation. (Work in progress.) URL: http://www.w3.org/TR/2014/CR-html5-20140204/

¶ **[MEDIAQ]**
Florian Rivoal. Media Queries. 19 June 2012. W3C Recommendation. URL: http://www.w3.org/TR/2012/REC-css3-mediaqueries-20120619/

§ Index

A, **6**

An+B, **6**

<an+b>, **6.2**

are a valid escape, **4.3.8**

ASCII case-insensitive, **5.2**

<at-keyword-token>, **4**

at-rule, **5**

B, **6**

<bad-string-token>, **4**

<bad-url-token>, **4**

<CDC-token>, **4**

<CDO-token>, **4**

@charset, **8.2**

check if three code points would start an identifier, **4.3.9**

check if three code points would start a number, **4.3.10**

check if two code points are a valid escape, **4.3.8**

code point, **4.2**

<colon-token>, **4**

<column-token>, **4**

<comma-token>, **4**

component value, **5**

consume a component value, **5.4.6**

consume a declaration, **5.4.5**

consume a function, **5.4.8**

consume a list of declarations, **5.4.4**

consume a list of rules, **5.4.1**

consume a name, **4.3.11**

consume an at-rule, **5.4.2**

consume an escaped code point, **4.3.7**

consume an ident-like token, **4.3.3**

consume a number, **4.3.12**

consume a numeric token, **4.3.2**

consume a qualified rule, **5.4.3**

consume a simple block, **5.4.7**

consume a string token, **4.3.4**

consume a token, **4.3.1**

consume a unicode-range token, **4.3.6**

consume a url token, **4.3.5**

consume the next input token, **5.2**

consume the remnants of a bad url, **4.3.14**

convert a string to a number, **4.3.13**

current input code point, **4.2**

current input token, **5.2**

<dash-match-token>, **4**

<dashndashdigit-ident>, **6.2**

declaration, **5**

<declaration-list>, **7.1**

decode, **3.2**

<delim-token>, **4**

determine the fallback encoding, **3.2**

digit, **4.2**

<dimension-token>, **4**

ending token, **5.4.7**

end of the range, **4**

environment encoding, **3.2**

EOF code point, **4.2**

<EOF-token>, **5.2**

escaping, **2.1**

function, **5**

<function-token>, **4**

get an encoding, **3.2**

<hash-token>, **4**

hex digit, **4.2**

identifier, **4.2**

<ident-token>, **4**

<include-match-token>, **4**

<integer>, **6.2**

letter, **4.2**

lowercase letter, **4.2**

maximum allowed code point, **4.2**

name code point, **4.2**

name-start code point, **4.2**

<ndashdigit-dimension>, **6.2**

<ndashdigit-ident>, **6.2**

<ndash-dimension>, **6.2**

<n-dimension>, **6.2**

newline, **4.2**

next input code point, **4.2**

next input token, **5.2**

non-ASCII code point, **4.2**

non-printable code point, **4.2**

<number-token>, **4**

parse a component value, **5.3.6**

parse a CSS stylesheet, **8**

parse a declaration, **5.3.4**

parse a list of component values, **5.3.7**

parse a list of declarations, **5.3.5**

parse a list of rules, **5.3.2**

parse a rule, **5.3.3**

parse a stylesheet, **5.3.1**

parse error, **3**

<percentage-token>, **4**

<prefix-match-token>, **4**

preserved tokens, **5**

qualified rule, **5**

reconsume the current input code point, **4.2**

reconsume the current input token, **5.2**

<rule-list>, **7.1**

<semicolon-token>, **4**

<signed-integer>, **6.2**

<signless-integer>, **6.2**

simple block, **5**

start of the range, **4**

starts with an identifier, **4.3.9**

starts with a number, **4.3.10**

starts with a valid escape, **4.3.8**

start with an identifier, **4.3.9**

start with a number, **4.3.10**

<string-token>, **4**

style rule, **8.1**

<stylesheet>, **7.1**

<substring-match-token>, **4**

<suffix-match-token>, **4**

surrogate code point, **4.2**

<{-token>, **4**

<)-token>, **4**

<}-token>, **4**

<]-token>, **4**

<(-token>, **4**

<[-token>, **4**

top-level flag, **5.4.1**

<unicode-range-token>, **4**

uppercase letter, **4.2**

<url-token>, **4**

whitespace, **4.2**

<whitespace-token>, **4**

would start an identifier, **4.3.9**

would start a number, **4.3.10**

§ Property index

No properties defined.