Selectors Level 4

W3C Working Draft, 11 November 2022

▼ **More details about this document**

This version:

https://www.w3.org/TR/2022/WD-selectors-4-20221111/

Latest published version:

https://www.w3.org/TR/selectors-4/

Editor's Draft:

https://drafts.csswg.org/selectors/

Previous Versions:

https://www.w3.org/TR/2022/WD-selectors-4-20220507/
https://www.w3.org/TR/2018/WD-selectors-4-20181121/
https://www.w3.org/TR/2018/WD-selectors-4-20180202/
https://www.w3.org/TR/2018/WD-selectors-4-20180201/
https://www.w3.org/TR/2013/WD-selectors4-20130502/
https://www.w3.org/TR/2012/WD-selectors4-20120823/
https://www.w3.org/TR/2011/WD-selectors4-20110929/

History:

https://www.w3.org/standards/history/selectors-4

Test Suite:

http://test.csswg.org/suites/selectors-4_dev/nightly-unstable/

Feedback:

CSSWG Issues Repository
Inline In Spec

Editors:

Elika J. Etemad / fantasai (Invited Expert)
Tab Atkins Jr. (Google)

Former Editors:

Tantek Çelik
Daniel Glazman
Ian Hickson
Peter Linss
John Williams

Suggest an Edit for this Spec:

GitHub Editor

rules apply.

Abstract

Selectors are patterns that match against elements in a tree, and as such form one of several technologies that can be used to select nodes in a document. Selectors have been optimized for use with HTML and XML, and are designed to be usable in performance-critical code. They are a core component of CSS (Cascading Style Sheets), which uses Selectors to bind style properties to elements in the document. Selectors Level 4 describes the selectors that already exist in [SELECT], and further introduces new selectors for CSS and other languages that may need them.

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

Status of this document

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

This document was published by the CSS Working Group as a **Working Draft** using the Recommendation track. Publication as a Working Draft does not imply endorsement by W3C and its Members.

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.

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

This document is governed by the 2 November 2021 W3C Process Document.

This document was produced by a group operating under the 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.

The following features are at-risk, and may be dropped during the CR period:

- the column combinator

- the ‘:read-write’ pseudo-class
- the ‘:has()’ pseudo-class

“At-risk” is a W3C Process term-of-art, and does not necessarily imply that the feature is in danger of being dropped or delayed. It means that the WG believes the feature may have difficulty being interoperably implemented in a timely manner, and marking it as such allows the WG to drop the feature if necessary when transitioning to the Proposed Rec stage, without having to publish a new Candidate Rec without the feature first.

Table of Contents

§ 1. Introduction

This section is not normative.

A selector is a boolean predicate that takes an element in a tree structure and tests whether the element matches the selector or not.

These expressions may be used for many things:

- directly on an element to test whether it matches some criteria, such as in the `element.matches()` function defined in [DOM]
- applied to an entire tree of elements to filter it into a set of elements that match the criteria, such as in the `document.querySelectorAll()` function defined in [DOM] or the selector of a CSS style rule.
- used "in reverse" to generate markup that would match a given selector, such as in HAML or Emmet.

Selectors Levels 1, 2, and 3 are defined as the subsets of selector functionality defined in the CSS1, CSS2.1, and Selectors Level 3 specifications, respectively. This module defines Selectors Level 4.

§ 1.1. Module Interactions

This module replaces the definitions of and extends the set of selectors defined for CSS in [SELECT] and [CSS21].

Pseudo-element selectors, which define abstract elements in a rendering tree, are not part of this specification: their generic syntax is described here, but, due to their close integration with the rendering model and irrelevance to other uses such as DOM queries, they will be defined in other

modules.

§ 2. Selectors Overview

This section is non-normative, as it merely summarizes the following sections.

A selector represents a structure. This structure can be used as a condition (e.g. in a CSS rule) that determines which elements a selector matches in the document tree, or as a flat description of the HTML or XML fragment corresponding to that structure.

Selectors may range from simple element names to rich contextual representations.

The following table summarizes the Selector syntax:

Pattern	Represents	Section	Level
`*`	any element	§ 5.2 Universal selector	2
`E`	an element of type E	§ 5.1 Type (tag name) selector	1
`E:not(`*`s1, s2, …`*`)`	an E element that does not match either compound selector *s1* or compound selector *s2*	§ 4.3 The Negation (Matches-None) Pseudo-class: :not()	3/4
`E:is(`*`s1, s2, …`*`)`	an E element that matches compound selector *s1* and/or compound selector *s2*	§ 4.2 The Matches-Any Pseudo-class: :is()	4
`E:where(`*`s1, s2, …`*`)`	an E element that matches compound selector *s1* and/or compound selector *s2* but contributes no specificity.	§ 4.4 The Specificity-adjustment Pseudo-class: :where()	4
`E:has(`*`rs1, rs2, …`*`)`	an E element, if there exists an element that matches either of the relative selectors *rs1* or *rs2*, when evaluated with E as the anchor elements	§ 4.5 The Relational Pseudo-class: :has()	4
`E.warning`	an E element belonging to the class `warning` (the document language specifies how class is determined).	§ 6.6 Class selectors	1
`E#myid`	an E element with ID equal to `myid`.	§ 6.7 ID selectors	1

`E[foo]`	an E element with a `foo` attribute	§ 6.1 Attribute presence and value selectors	2
`E[foo="bar"]`	an E element whose `foo` attribute value is exactly equal to `bar`	§ 6.1 Attribute presence and value selectors	2
`E[foo="bar" i]`	an E element whose `foo` attribute value is exactly equal to any (ASCII-range) case-permutation of `bar`	§ 6.3 Case-sensitivity	4
`E[foo="bar" s]`	an E element whose `foo` attribute value is identical to `bar`	§ 6.3 Case-sensitivity	4
`E[foo~="bar"]`	an E element whose `foo` attribute value is a list of whitespace-separated values, one of which is exactly equal to `bar`	§ 6.1 Attribute presence and value selectors	2
`E[foo^="bar"]`	an E element whose `foo` attribute value begins exactly with the string `bar`	§ 6.2 Substring matching attribute selectors	3
`E[foo$="bar"]`	an E element whose `foo` attribute value ends exactly with the string `bar`	§ 6.2 Substring matching attribute selectors	3
`E[foo*="bar"]`	an E element whose `foo` attribute value contains the substring `bar`	§ 6.2 Substring matching attribute selectors	3
`E[foo|="en"]`	an E element whose `foo` attribute value is a hyphen-separated list of values	§ 6.1 Attribute presence and value selectors	2

	beginning with en		
E:dir(ltr)	an element of type E with left-to-right directionality (the document language specifies how directionality is determined)	§ 7.1 The Directionality Pseudo-class: :dir()	4
E:lang(zh, "*-hant")	an element of type E tagged as being either in Chinese (any dialect or writing system) or otherwise written with traditional Chinese characters	§ 7.2 The Language Pseudo-class: :lang()	2/4
E:any-link	an E element being the source anchor of a hyperlink	§ 8.1 The Hyperlink Pseudo-class: :any-link	4
E:link	an E element being the source anchor of a hyperlink of which the target is not yet visited	§ 8.2 The Link History Pseudo-classes: :link and :visited	1
E:visited	an E element being the source anchor of a hyperlink of which the target is already visited	§ 8.2 The Link History Pseudo-classes: :link and :visited	1
E:local-link	an E element being the source anchor of a hyperlink targeting the current URL	§ 8.3 The Local Link Pseudo-class: :local-link	4
E:target	an E element being the target of the current URL	§ 8.4 The Target Pseudo-class: :target	3

`E:target-within`	an E element that is the target of the current URL or contains an element that does.	§ 8.5 The Target Container Pseudo-class: :target-within	4
`E:scope`	an E element being a scoping root	§ 8.6 The Reference Element Pseudo-class: :scope	4
`E:current`	an E element that is currently presented in a time-dimensional canvas	§ 10.1 The Current-element Pseudo-class: :current	4
`E:current(s)`	an E element that is the deepest ':current' element that matches selector *s*	§ 10.1 The Current-element Pseudo-class: :current	4
`E:past`	an E element that is in the past in a time-dimensional canvas	§ 10.2 The Past-element Pseudo-class: :past	4
`E:future`	an E element that is in the future in a time-dimensional canvas	§ 10.3 The Future-element Pseudo-class: :future	4
`E:active`	an E element that is in an activated state	§ 9.2 The Activation Pseudo-class: :active	1
`E:hover`	an E element that is under the cursor, or that has a descendant under the cursor	§ 9.1 The Pointer Hover Pseudo-class: :hover	2
`E:focus`	an E element that has user input focus	§ 9.3 The Input Focus Pseudo-class: :focus	2

`E:focus-within`	an E element that has user input focus or contains an element that has input focus.	§ 9.5 The Focus Container Pseudo-class: :focus-within	4
`E:focus-visible`	an E element that has user input focus, and the UA has determined that a focus ring or other indicator should be drawn for that element	§ 9.4 The Focus-Indicated Pseudo-class: :focus-visible	4
`E:enabled` `E:disabled`	a user interface element E that is enabled or disabled, respectively	§ 13.1.1 The :enabled and :disabled Pseudo-classes	3
`E:read-write` `E:read-only`	a user interface element E that is user alterable, or not	§ 13.1.2 The Mutability Pseudo-classes: :read-only and :read-write	3-UI/4
`E:placeholder-shown`	an input control currently showing placeholder text	§ 13.1.3 The Placeholder-shown Pseudo-class: :placeholder-shown	3-UI/4
`E:default`	a user interface element E that is the default item in a group of related choices	§ 13.1.5 The Default-option Pseudo-class: :default	3-UI/4
`E:checked`	a user interface element E that is checked/selected (for instance a radio-button or checkbox)	§ 13.2.1 The Selected-option Pseudo-class: :checked	3

`E:indeterminate`	a user interface element E that is in an indeterminate state (neither checked nor unchecked)	§ 13.2.2 The Indeterminate-value Pseudo-class: :indeterminate	4
`E:valid` `E:invalid`	a user-input element E that meets, or doesn't, its data validity semantics	§ 13.3.2 The Validity Pseudo-classes: :valid and :invalid	3-UI/4
`E:in-range` `E:out-of-range`	a user-input element E whose value is in-range/out-of-range	§ 13.3.3 The Range Pseudo-classes: :in-range and :out-of-range	3-UI/4
`E:required` `E:optional`	a user-input element E that requires/does not require input	§ 13.3.4 The Optionality Pseudo-classes: :required and :optional	3-UI/4
`E:blank`	a user-input element E whose value is blank (empty/missing)	§ 13.3.1 The Empty-Value Pseudo-class: :blank	4
`E:user-invalid`	a user-altered user-input element E with incorrect input (invalid, out-of-range, omitted-but-required)	§ 13.3.5 The User-interaction Pseudo-classes: :user-valid and :user-invalid	4
`E:root`	an E element, root of the document	§ 14.1 :root pseudo-class	3
`E:empty`	an E element that has no children (neither elements nor text) except perhaps white space	§ 14.2 :empty pseudo-class	3
`E:nth-child(`*n* `[of` *S*`]?)`	an E element, the *n*-th child of its parent	§ 14.3.1 :nth-child() pseudo-	3/4

	matching *S*	class	
`E:nth-last-child(n [of S]?)`	an E element, the *n*-th child of its parent matching *S*, counting from the last one	§ 14.3.2 :nth-last-child() pseudo-class	3/4
`E:first-child`	an E element, first child of its parent	§ 14.3.3 :first-child pseudo-class	2
`E:last-child`	an E element, last child of its parent	§ 14.3.4 :last-child pseudo-class	3
`E:only-child`	an E element, only child of its parent	§ 14.3.5 :only-child pseudo-class	3
`E:nth-of-type(n)`	an E element, the *n*-th sibling of its type	§ 14.4.1 :nth-of-type() pseudo-class	3
`E:nth-last-of-type(n)`	an E element, the *n*-th sibling of its type, counting from the last one	§ 14.4.2 :nth-last-of-type() pseudo-class	3
`E:first-of-type`	an E element, first sibling of its type	§ 14.4.3 :first-of-type pseudo-class	3
`E:last-of-type`	an E element, last sibling of its type	§ 14.4.4 :last-of-type pseudo-class	3
`E:only-of-type`	an E element, only sibling of its type	§ 14.4.5 :only-of-type pseudo-class	3
`E F`	an F element descendant of an E element	§ 15.1 Descendant combinator ( )	1
`E > F`	an F element child of	§ 15.2 Child	2

E > F	an E element	combinator (>)	2
E + F	an F element immediately preceded by an E element	§ 15.3 Next-sibling combinator (+)	2
E ~ F	an F element preceded by an E element	§ 15.4 Subsequent-sibling combinator (~)	3
F || E	an E element that represents a cell in a grid/table belonging to a column represented by an element F	§ 16.1 Column combinator (||)	4
E:nth-col(*n*)	an E element that represents a cell belonging to the *n*th column in a grid/table	§ 16.2 :nth-col() pseudo-class	4
E:nth-last-col(*n*)	an E element that represents a cell belonging to the *n*th column in a grid/table, counting from the last one	§ 16.3 :nth-last-col() pseudo-class	4

Note: Some Level 4 selectors (noted above as "3-UI") were introduced in [CSS3UI].

§ 3. Selector Syntax and Structure

§ 3.1. Structure and Terminology

A ***selector*** represents a particular pattern of element(s) in a tree structure. The term selector can refer to a simple selector, compound selector, complex selector, or selector list. The ***subject of a selector*** is any element that selector is defined to be about; that is, any element ***matching*** that selector.

A ***simple selector*** is a single condition on an element. A type selector, universal selector, attribute

selector, class selector, ID selector, or pseudo-class is a simple selector. (It is represented by <simple-selector> in the selectors grammar.) A given element is said to match a simple selector when that simple selector, as defined in this specification and in accordance with the document language, accurately describes the element.

A ***compound selector*** is a sequence of simple selectors that are not separated by a combinator, and represents a set of simultaneous conditions on a single element. If it contains a type selector or universal selector, that selector must come first in the sequence. Only one type selector or universal selector is allowed in the sequence. (A compound selector is represented by <compound-selector> in the selectors grammar.) A given element is said to match a compound selector when it matches all simple selectors in the compound selector.

> Note: As whitespace represents the descendant combinator, no whitespace is allowed between the simple selectors in a compound selector.

A ***combinator*** is a condition of relationship between two elements represented by the compound selectors on either side. Combinators in Selectors Level 4 include: the descendant combinator (white space), the child combinator (U+003E, >), the next-sibling combinator (U+002B, +), and the subsequent-sibling combinator (U+007E, ~). Two given elements are said to match a combinator when the condition of relationship between these elements is true.

A ***complex selector*** is a sequence of one or more compound selectors separated by combinators. It represents a set of simultaneous conditions on a set of elements in the particular relationships described by its combinators. (Complex selectors are represented by <complex-selector> in the selectors grammar.) A given element is said to match a complex selector when there exists a list of elements, each matching a corresponding compound selector in the complex selector, with each pair of elements consecutive in the list matching the combinator between their corresponding compound selectors, and with the last element being the given element.

> Note: Thus, a selector consisting of a single compound selector matches any element satisfying the requirements of its constituent simple selectors. Prepending another compound selector and a combinator to a sequence imposes additional matching constraints, such that the subjects of a complex selector are always a subset of the elements represented by its last compound selector.

A ***list of simple/compound/complex selectors*** is a comma-separated list of simple, compound, or complex selectors. This is also called just a ***selector list*** when the type is either unimportant or specified in the surrounding prose; if the type is important and unspecified, it defaults to meaning a list of complex selectors. (See § 4.1 Selector Lists for additional information on selector lists and the various <*-selector-list> productions in the grammar for their formal syntax.) A given element is said to match a selector list when it matches any (at least one) of the selectors in that selector list.

> ISSUE 1 Pseudo-elements aren't handled here, and should be.

§ 3.2. Data Model

Selectors are evaluated against an element tree such as the DOM. [DOM] Within this specification, this may be referred to as the "document tree" or "source document".

Each element may have any of the following five aspects, which can be selected against, all of which are matched as strings:

- The element's type (also known as its tag name).
- The element's namespace.
- An ID.
- Classes (named groups) to which it belongs.
- Attributes, which are name-value pairs.

While individual elements may lack any of the above features, some elements are ***featureless***. A featureless element does not match any selector at all, except those it is explicitly defined to match. If a given selector *is* allowed to match a featureless element, it must do so while ignoring the default namespace. [CSS3NAMESPACE]

> EXAMPLE 1
> For example, the shadow host in a shadow tree is featureless, and can't be matched by *any* pseudo-class except for ':host' and ':host-context()'.)

Many of the selectors depend on the semantics of the ***document language*** (i.e. the language and semantics of the document tree) and/or the semantics of the ***host language*** (i.e. the language that is using selectors syntax). For example, the ':lang()' selector depends on the document language (e.g. HTML) to define how an element is associated with a language. As a slightly different example, the '::first-line' pseudo-element depends on the host language (e.g. CSS) to define what a '::first-line' pseudo-element represents and what it can do.

§ 3.3. Scoped Selectors

Some host applications may choose to ***scope*** selectors to a particular subtree or fragment of the document, The root of the scoping subtree is called the ***scoping root***.

When a selector is scoped, it matches an element only if the element is a descendant of the scoping root. (The rest of the selector can match unrestricted; it's only the final matched elements that must be within the scope.)

EXAMPLE 2

For example, the `querySelector()` method defined in [DOM] allows the author to evaluate a scoped selector relative to the element it's called on.

A call like `widget.querySelector("a")` will thus only find `<a>` elements inside of the `widget` element, ignoring any other `<a>`s that might be scattered throughout the document.

§ 3.4. Relative Selectors

Certain contexts may accept ***relative selectors***, which are a shorthand for selectors that represent elements relative to one or more ***relative selector anchor elements***. Relative selectors begin with a combinator, with a selector representing the anchor element implied at the start of the selector. (If no combinator is present, the descendant combinator is implied.)

Relative selectors are represented by <relative-selector> in the selectors grammar, and lists of them by <relative-selector-list>.

§ 3.5. Pseudo-classes

Pseudo-classes are simple selectors that permit selection based on information that lies outside of the document tree or that can be awkward or impossible to express using the other simple selectors. They can also be dynamic, in the sense that an element can acquire or lose a pseudo-class while a user interacts with the document, without the document itself changing. Pseudo-classes do not appear in or modify the document source or document tree.

The syntax of a pseudo-class consists of a ":" (U+003A COLON) followed by the name of the pseudo-class as a CSS identifier, and, in the case of a ***functional pseudo-class***, a pair of parentheses containing its arguments.

EXAMPLE 3

For example, ‘:valid’ is a regular pseudo-class, and ‘:lang()’ is a functional pseudo-class.

Like all CSS keywords, pseudo-class names are ASCII case-insensitive. No white space is allowed between the colon and the name of the pseudo-class, nor, as usual for CSS syntax, between a functional pseudo-class's name and its opening parenthesis (which thus form a CSS function token). Also as usual, white space is allowed around the arguments inside the parentheses of a functional pseudo-class unless otherwise specified.

Like other simple selectors, pseudo-classes are allowed in all compound selectors contained in a selector, and must follow the type selector or universal selector, if present.

Note: Some pseudo-classes are mutually exclusive (such that a compound selector containing them, while valid, will never match anything), while others can apply simultaneously to the same element.

§ 3.6. Pseudo-elements

Similar to how certain pseudo-classes represent additional state information not directly present in the document tree, a ***pseudo-element*** represents an *element* not directly present in the document tree. They are used to create abstractions about the document tree beyond those provided by the document tree. For example, pseudo-elements can be used to select portions of the document that do not correspond to a document-language element (including such ranges as don't align to element boundaries or fit within its tree structure); that represent content not in the document tree or in an alternate projection of the document tree; or that rely on information provided by styling, layout, user interaction, and other processes that are not reflected in the document tree.

EXAMPLE 4

For instance, document languages do not offer mechanisms to access the first letter or first line of an element's content, but there exist pseudo-elements (':::first-letter' and ':::first-line') that allow those things to be styled. Notice especially that in the case of ':::first-line', which portion of content is represented by the pseudo-element depends on layout information that cannot be inferred from the document tree.

Pseudo-elements can also represent content that doesn't exist in the source document at all, such as the ':::before' and ':::after' pseudo-elements which allow additional content to be inserted before or after the contents of any element.

Like pseudo-classes pseudo-elements do not appear in or modify the document source or document tree. Accordingly, they also do not affect the interpretation of structural pseudo-classes or other selectors pertaining to their originating element or its tree.

The host language defines which pseudo-elements exist, their type, and their abilities. Pseudo-elements that exist in CSS are defined in [CSS21] (Level 2), [SELECT] (Level 3), and [CSS-PSEUDO-4] (Level 4).

§ 3.6.1. Syntax

The syntax of a pseudo-element is "::" (two U+003A COLON characters) followed by the name of the pseudo-element as an identifier. Pseudo-element names are ASCII case-insensitive. No white space is allowed between the two colons, or between the colons and the name.

Because CSS Level 1 and CSS Level 2 conflated pseudo-elements and pseudo-classes by sharing a single-colon syntax for both, user agents must also accept the previous one-colon notation for the Level 1 & 2 pseudo-elements (‘::before’, ‘::after’, ‘::first-line’, and ‘::first-letter’). This compatibility notation is not allowed for any other pseudo-elements. However, as this syntax is deprecated, authors should use the Level 3+ double-colon syntax for these pseudo-elements.

Pseudo-elements are featureless, and so can’t be matched by any other selector.

§ 3.6.2. Binding to the Document Tree

Pseudo-elements do not exist independently in the tree: they are always bound to another element on the page, called their ***originating element***. Syntactically, a pseudo-element immediately follows the compound selector representing its originating element. If this compound selector is omitted, it is assumed to be the universal selector ‘*’.

> EXAMPLE 5
> For example, in the selector ‘div a::before’, the ‘a’ elements matched by the selector are the originating elements for the ‘::before’ pseudo-elements attached to them.
>
> The selector ‘::first-line’ is equivalent to ‘*::first-line’, which selects the ‘::first-line’ pseudo-element on *every* element in the document.

When a pseudo-element is encountered in a selector, the part of the selector before the pseudo-element selects the originating element for the pseudo-element; the part of the selector after it, if any, applies to the pseudo-element itself. (See below.)

§ 3.6.3. Pseudo-classing Pseudo-elements

A pseudo-element may be immediately followed by any combination of the user action pseudo-classes, in which case the pseudo-element is represented only when it is in the corresponding state. Whether these pseudo-classes can match on the pseudo-element depends on the pseudo-class and pseudo-element’s definitions: unless otherwise-specified, none of these pseudo-classes will match on the pseudo-element.

> ISSUE 2 Clarify that ‘:not()’ and ‘:is()’ can be used when containing above-mentioned pseudos.

EXAMPLE 6

For example, since the ‘:hover’ pseudo-class specifies that it can apply to any pseudo-element, ‘::first-line:hover’ will match when the first line is hovered. However, since neither ‘:focus’ nor ‘::first-line’ define that ‘:focus’ can apply to ‘::first-line’, the selector ‘::first-line:focus’ will never match anything.

ISSUE 3 Does ‘::first-line:not(:focus)’ match anything?

Notice that ‘::first-line:hover’ is very different from ‘:hover::first-line’, which matches the first line of any originating element that is hovered! For example, ‘:hover::first-line’ also matches the first line of a paragraph when the second line of the paragraph is hovered, whereas ‘::first-line:hover’ only matches if the first line itself is hovered.

Note: Note that, unless otherwise specified in a future specification, pseudo-classes other than the user action pseudo-classes are not valid when compounded to a pseudo-element; so, for example, ‘::before:first-child’ is an invalid selector.

§ 3.6.4. Sub-pseudo-elements

Some pseudo-elements are able to be the originating element of other pseudo-elements, which are defined as the ***sub-pseudo-elements*** of this ***originating pseudo-element***. For example, when ‘::before’ is given a ‘list-item’ display type, it becomes the originating pseudo-element of its ‘::before::marker’ sub-pseudo-element.

Where disambiguation is needed, the term ***ultimate originating element*** refers to the real (non-pseudo) element from which a pseudo-element originates.

Unless the corresponding sub-pseudo-element is explicitly defined to exist in another specification, pseudo-element selectors are not valid when compounded to another pseudo-element selector. So, for example, ‘::before::before’ is an invalid selector, but ‘::before::marker’ is valid (in implementations that support the ‘::before::marker’ sub-pseudo-element).

§ 3.6.5. Internal Structure

Some pseudo-elements are defined to have internal structure. These pseudo-elements may be followed by child/descendant combinators to express those relationships. Selectors containing combinators after the pseudo-element are otherwise invalid.

EXAMPLE 7

For example, '::first-letter + span' and '::first-letter em' are invalid selectors. However, if a new '::shadow' pseudo-element were defined to have internal structure, '::shadow > p' would be a valid selector.

Note: A future specification may expand the capabilities of existing pseudo-elements, so some of these currently-invalid selectors (e.g. '::first-line :any-link') may become valid in the future.

The children of such pseudo-elements can simultaneously be children of other elements, too. However, at least in CSS, their rendering must be defined so as to maintain the tree-ness of the box tree.

EXAMPLE 8

For example, the '::slotted' pseudo-element treats elements distributed to it as its children. This means that, given the following fragment:

```
<div>
  <span>foo</span>
  <"shadow root">
    <content></content>
  </"shadow root">
</div>
```

the selectors 'div > span' and 'div::shadow ::slotted > span' select the same element via different paths.

However, when rendered, the `<span>` element generates boxes as if it were the child of the `<content>` element, rather than the `<div>` element, so the tree structure of the box tree is maintained.

§ 3.7. Characters and case sensitivity

All Selectors syntax is ASCII case-insensitive (i.e. [a-z] and [A-Z] are equivalent), except for the parts that are not under the control of Selectors: specifically, the case-sensitivity of document language element names, attribute names, and attribute values depends on the document language.

EXAMPLE 9

For example, in HTML, element and attribute names are ASCII case-insensitive, but in XML, they are case-sensitive.

Case sensitivity of namespace prefixes is defined in [CSS3NAMESPACE]. Case sensitivity of

language ranges is defined in the ':lang()' section.

White space in Selectors consists of the code points SPACE (U+0020), TAB (U+0009), LINE FEED (U+000A), CARRIAGE RETURN (U+000D), and FORM FEED (U+000C). Other space-like code points, such as EM SPACE (U+2003) and IDEOGRAPHIC SPACE (U+3000), are never considered syntactic white space.

Code points in Selectors can be escaped with a backslash according to the same escaping rules as CSS. [CSS21] Note that escaping a code point "cancels out" any special meaning it may have in Selectors. For example, the selector '#foo>a' contains a combinator, but '#foo\>a' instead selects an element with the id `foo>a`.

§ 3.8. Declaring Namespace Prefixes

Certain selectors support namespace prefixes. The mechanism by which namespace prefixes are ***declared*** should be specified by the language that uses Selectors. If the language does not specify a namespace prefix declaration mechanism, then no prefixes are declared. In CSS, namespace prefixes are declared with the '@namespace' rule. [CSS3NAMESPACE]

§ 3.9. Invalid Selectors and Error Handling

User agents must observe the rules for handling ***invalid selectors***:

- a parsing error in a selector, e.g. an unrecognized token or a token which is not allowed at the current parsing point (see overall § 18 Grammar and per-selector syntax definitions), causes that selector to be invalid.
- a simple selector containing an undeclared namespace prefix is invalid
- a selector containing an invalid simple selector, an invalid combinator or an invalid token is invalid.
- a selector list containing an invalid selector is invalid.
- an empty selector, i.e. one that contains no compound selector, is invalid.

> Note: Consistent with CSS's forwards-compatible parsing principle, UAs *must* treat as invalid any pseudo-classes, pseudo-elements, combinators, or other syntactic constructs for which they have no usable level of support. See Partial implementations.

An invalid selector represents, and therefore matches, nothing.

§ 3.10. Legacy Aliases

Some selectors have a ***legacy selector alias***. This is a name which, at parse time, is converted to the standard name (and thus does not appear anywhere in any object model representing the selector).

§ 4. Logical Combinations

§ 4.1. Selector Lists

A comma-separated list of selectors represents the union of all elements selected by each of the individual selectors in the selector list. (A comma is U+002C.) For example, in CSS when several selectors share the same declarations, they may be grouped into a comma-separated list. White space may appear before and/or after the comma.

> EXAMPLE 10
> CSS example: In this example, we condense three rules with identical declarations into one. Thus,
>
> ```
> h1 { font-family: sans-serif }
> h2 { font-family: sans-serif }
> h3 { font-family: sans-serif }
> ```
>
> is equivalent to:
>
> ```
> h1, h2, h3 { font-family: sans-serif }
> ```

Warning: the equivalence is true in this example because all the selectors are valid selectors. If just one of these selectors were invalid, the entire selector list would be invalid. This would invalidate the rule for all three heading elements, whereas in the former case only one of the three individual heading rules would be invalidated.

> EXAMPLE 11
> Invalid CSS example:
>
> ```
> h1 { font-family: sans-serif }
> h2..foo { font-family: sans-serif }
> h3 { font-family: sans-serif }
> ```
>
> is not equivalent to:
>
> ```
> h1, h2..foo, h3 { font-family: sans-serif }
> ```
>
> because the above selector (‘h1, h2..foo, h3’) is entirely invalid and the entire style rule is dropped. (When the selectors are not grouped, only the rule for ‘h2..foo’ is dropped.)

§ 4.2. The Matches-Any Pseudo-class: ':is()'

The matches-any pseudo-class, ***':is()'***, is a functional pseudo-class taking a <forgiving-selector-list> as its sole argument.

If the argument, after parsing, is an empty list, the pseudo-class is valid but matches nothing. Otherwise, the pseudo-class matches any element that matches any of the selectors in the list.

> Note: The specificity of the ':is()' pseudo-class is replaced by the specificity of its most specific argument. Thus, a selector written with ':is()' does not necessarily have equivalent specificity to the equivalent selector written without ':is()' For example, if we have ':is(ul, ol, .list) > [hidden]' and 'ul > [hidden], ol > [hidden], .list > [hidden]' a '[hidden]' child of an `<ol>` matches the first selector with a specificity of (0,2,0) whereas it matches the second selector with a specificity of (0,1,1). See § 17 Calculating a selector's specificity.

Pseudo-elements cannot be represented by the matches-any pseudo-class; they are not valid within ':is()'.

Default namespace declarations do not affect the compound selector representing the subject of any selector within a ':is()' pseudo-class, unless that compound selector contains an explicit universal selector or type selector.

> EXAMPLE 12
> For example, the following selector matches any element that is being hovered or focused, regardless of its namespace. In particular, it is not limited to only matching elements in the default namespace that are being hovered or focused.
>
> ```
> *|*:is(:hover, :focus)
> ```
>
> The following selector, however, represents only hovered or focused elements that are in the default namespace, because it uses an explicit universal selector within the ':is()' notation:
>
> ```
> *|*:is(*:hover, *:focus)
> ```

As previous drafts of this specification used the name ***':matches()'*** for this pseudo-class, UAs may additionally implement this obsolete name as a legacy selector alias for ':is()' if needed for backwards-compatibility.

§ 4.3. The Negation (Matches-None) Pseudo-class: ':not()'

The negation pseudo-class, ***':not()'***, is a functional pseudo-class taking a selector list as an argument. It represents an element that is not represented by its argument.

Note: In Selectors Level 3, only a single simple selector was allowed as the argument to ':not()'.

Note: The specificity of the ':not()' pseudo-class is replaced by the specificity of the most specific selector in its argument; thus it has the exact behavior of ':not(:is(argument))'. See § 17 Calculating a selector's specificity.

Pseudo-elements cannot be represented by the negation pseudo-class; they are not valid within ':not()'.

EXAMPLE 13

For example, the following selector matches all `<button>` elements in an HTML document that are not disabled.

```
button:not([DISABLED])
```

The following selector represents all but FOO elements.

```
*:not(FOO)
```

The following compound selector represents all HTML elements except links.

```
html|*:not(:link):not(:visited)
```

As with ':is()', default namespace declarations do not affect the compound selector representing the subject of any selector within a ':not()' pseudo-class, unless that compound selector contains an explicit universal selector or type selector. (See ':is()' for examples.)

Note: The ':not()' pseudo-class allows useless selectors to be written. For instance ':not(*|*)', which represents no element at all, or 'div:not(span)', which is equivalent to 'div' but with a higher specificity.

§ 4.4. The Specificity-adjustment Pseudo-class: ':where()'

The Specificity-adjustment pseudo-class, ***':where()'***, is a functional pseudo-class with the same syntax and functionality as ':is()'. Unlike ':is()', neither the ':where()' pseudo-class, nor any of its arguments, contribute to the specificity of the selector—its specificity is always zero.

This is useful for introducing filters in a selector while keeping the associated style declarations easy to override.

EXAMPLE 14

Below is a common example where the specificity heuristic fails to match author expectations:

```
a:not(:hover) {
  text-decoration: none;
}

nav a {
  /* Has no effect */
  text-decoration: underline;
}
```

However, by using ‘:where()’ the author can explicitly declare their intent:

```
a:where(:not(:hover)) {
  text-decoration: none;
}

nav a {
  /* Works now! */
  text-decoration: underline;
}
```

Note: Future levels of Selectors may introduce an additional argument to explicitly set the specificity of that instance of the pseudo-class.

§ 4.5. The Relational Pseudo-class: ‘:has()’

The relational pseudo-class, ***‘:has()’***, is a functional pseudo-class taking a <forgiving-relative-selector-list> as an argument. It represents an element if any of the relative selectors would match at least one element when anchored against this element.

The ‘:has()’ pseudo-class cannot be nested; ‘:has()’ is not valid within ‘:has()’. Also, unless explicitly defined as a ***:has-allowed pseudo-element***, pseudo-elements are not valid selectors within ‘:has()’. (This specification does not define any :has-allowed pseudo-elements, but other specifications may do so.)

Note: Pseudo-elements are generally excluded from ‘:has()’ because many of them exist conditionally, based on the styling of their ancestors, so allowing these to be queried by ‘:has()’ would introduce cycles.

EXAMPLE 15

For example, the following selector matches only `<a>` elements that contain an `<img>` child:

```
a:has(> img)
```

The following selector matches a `<dt>` element immediately followed by another `<dt>` element:

```
dt:has(+ dt)
```

The following selector matches `<section>` elements that don't contain any heading elements:

```
section:not(:has(h1, h2, h3, h4, h5, h6))
```

Note that ordering matters in the above selector. Swapping the nesting of the two pseudo-classes, like:

```
section:has(:not(h1, h2, h3, h4, h5, h6))
```

...would result in matching any `<section>` element which contains anything that's not a heading element.

§ 5. Elemental selectors

§ 5.1. Type (tag name) selector

A ***type selector*** is the name of a document language element type, and represents an instance of that element type in the document tree.

EXAMPLE 16

For example, the selector 'h1' represents an `<h1>` element in the document.

A type selector is written as a CSS qualified name: an identifier with an optional namespace prefix. [CSS3NAMESPACE] (See § 5.3 Namespaces in Elemental Selectors.)

§ 5.2. Universal selector

The ***universal selector*** is a special type selector, that represents an element of any element type.

It is written as a CSS qualified name with an asterisk (* U+002A) as the local name. Like a type selector, the universal selector can be qualified by a namespace, restricting it to only elements belonging to that namespace, and is affected by a default namespace as defined in § 5.3

Namespaces in Elemental Selectors.

Unless an element is featureless, the presence of a universal selector has no effect on whether the element matches the selector. (Featureless elements do not match any selector, including the universal selector.)

> EXAMPLE 17
>
> - '*[hreflang|=en]' and '[hreflang|=en]' are equivalent,
> - '*.warning' and '.warning' are equivalent,
> - '*#myid' and '#myid' are equivalent.

The universal selector follows the same syntax rules as other type selectors: only one can appear per compound selector, and it must be the first simple selector in the compound selector.

> Note: In some cases, adding a universal selector can make a selector easier to read, even though it has no effect on the matching behavior. For example, 'div :first-child' and 'div:first-child' are somewhat difficult to tell apart at a quick glance, but writing the former as 'div *:first-child' makes the difference obvious.

§ 5.3. Namespaces in Elemental Selectors

Type selectors and universal selectors allow an optional namespace component: a namespace prefix that has been previously declared may be prepended to the element name separated by the namespace separator "vertical bar" (| U+007C). (See, e.g., [XML-NAMES] for the use of namespaces in XML.) It has the following meaning in each form:

ns|E
: elements with name E in namespace ns

***|E**
: elements with name E in any namespace, including those without a namespace

|E
: elements with name E without a namespace

E
: if no default namespace has been declared for selectors, this is equivalent to *|E. Otherwise it is equivalent to ns|E where ns is the default namespace.

EXAMPLE 18

CSS examples:

```
@namespace foo url(http://www.example.com);
foo|h1 { color: blue }  /* first rule */
foo|* { color: yellow } /* second rule */
|h1 { color: red }      /* ...*/
*|h1 { color: green }
h1 { color: green }
```

The first rule (not counting the '@namespace' at-rule) will match only `<h1>` elements in the "http://www.example.com" namespace.

The second rule will match all elements in the "http://www.example.com" namespace.

The third rule will match only `<h1>` elements with no namespace.

The fourth rule will match `<h1>` elements in any namespace (including those without any namespace).

The last rule is equivalent to the fourth rule because no default namespace has been defined.

If a default namespace is declared, compound selectors without type selectors in them still only match elements in that default namespace.

EXAMPLE 19

For example, in the following style sheet:

```
@namespace url("http://example.com/foo");

.special { ... }
```

The '.special' selector only matches elements in the "http://example.com/foo" namespace, even though no reference to the type name (which is paired with the namespace in the DOM) appeared.

A type selector or universal selector containing a namespace prefix that has not been previously declared is an invalid selector.

§ 5.4. The Defined Pseudo-class: ':defined'

In some host languages, elements can have a distinction between being "defined"/"constructed" or not. The ***':defined'*** pseudo-class matches elements that are fully defined, as dictated by the host

language.

If the host language does not have this sort of distinction, all elements in it match ‘:defined’.

EXAMPLE 20
In HTML, all built-in elements are always considered to be defined, so the following example will always match:

```
p:defined { ... }
```

Custom elements, on the other hand, start out *un*defined, and only become defined when properly registered. This means the ‘:defined’ pseudo-class can be used to hide a custom element until it has been registered:

```
custom-element { visibility: hidden }

custom-element:defined { visibility: visible }
```

§ 6. Attribute selectors

Selectors allow the representation of an element’s attributes. When a selector is used as an expression to match against an element, an ***attribute selector*** must be considered to match an element if that element has an attribute that matches the attribute represented by the attribute selector.

ISSUE 4 Add comma-separated syntax for multiple-value matching? e.g. [rel ~= next, prev, up, first, last]

§ 6.1. Attribute presence and value selectors

CSS2 introduced four attribute selectors:

‘[att]’
Represents an element with the `att` attribute, whatever the value of the attribute.

‘[att=val]’
Represents an element with the `att` attribute whose value is exactly "val".

‘[att~=val]’
Represents an element with the `att` attribute whose value is a whitespace-separated list of words, one of which is exactly "val". If "val" contains whitespace, it will never represent anything (since the words are *separated* by spaces). Also if "val" is the empty string, it will

never represent anything.

‘[att|=val]’

Represents an element with the `att` attribute, its value either being exactly "val" or beginning with "val" immediately followed by "-" (U+002D). This is primarily intended to allow language subcode matches (e.g., the `hreflang` attribute on the `<a>` element in HTML) as described in BCP 47 ([BCP47]) or its successor. For `lang` (or `xml:lang`) language subcode matching, please see the ‘:lang()’ pseudo-class.

Attribute values must be <ident-token>s or <string-token>s. [CSS3SYN]

EXAMPLE 21

Examples:

The following attribute selector represents an `<h1>` element that carries the `title` attribute, whatever its value:

```
h1[title]
```

In the following example, the selector represents a `<span>` element whose `class` attribute has exactly the value "example":

```
span[class="example"]
```

Multiple attribute selectors can be used to represent several attributes of an element, or several conditions on the same attribute. Here, the selector represents a `<span>` element whose `hello` attribute has exactly the value "Cleveland" and whose `goodbye` attribute has exactly the value "Columbus":

```
span[hello="Cleveland"][goodbye="Columbus"]
```

The following CSS rules illustrate the differences between "=" and "~=". The first selector would match, for example, an `<a>` element with the value "copyright copyleft copyeditor" on a `rel` attribute. The second selector would only match an a element with an `href` attribute having the exact value "http://www.w3.org/".

```
a[rel~="copyright"] { ... }
a[href="http://www.w3.org/"] { ... }
```

The following selector represents an `<a>` element whose `hreflang` attribute is exactly "fr".

```
a[hreflang=fr]
```

The following selector represents an `<a>` element for which the value of the `hreflang` attribute begins with "en", including "en", "en-US", and "en-scouse":

```
a[hreflang|="en"]
```

The following selectors represent a `<DIALOGUE>` element whenever it has one of two different values for an attribute `character`:

```
DIALOGUE[character=romeo]
DIALOGUE[character=juliet]
```

§ 6.2. Substring matching attribute selectors

Three additional attribute selectors are provided for matching substrings in the value of an attribute:

'[att^=val]'
: Represents an element with the `att` attribute whose value begins with the prefix "val". If "val" is the empty string then the selector does not represent anything.

'[att$=val]'
: Represents an element with the `att` attribute whose value ends with the suffix "val". If "val" is the empty string then the selector does not represent anything.

'[att*=val]'
: Represents an element with the `att` attribute whose value contains at least one instance of the substring "val". If "val" is the empty string then the selector does not represent anything.

Attribute values must be <ident-token>s or <string-token>s.

> EXAMPLE 22
> Examples: The following selector represents an HTML `<object>` element, referencing an image:
>
> ```
> object[type^="image/"]
> ```
>
> The following selector represents an HTML `<a>` element with an `href` attribute whose value ends with ".html".
>
> ```
> a[href$=".html"]
> ```
>
> The following selector represents an HTML paragraph with a `title` attribute whose value contains the substring "hello"
>
> ```
> p[title*="hello"]
> ```

§ 6.3. Case-sensitivity

By default case-sensitivity of attribute names and values in selectors depends on the document language.

To match attribute values ASCII case-insensitively regardless of document language rules, the attribute selector may include the identifier `i` before the closing bracket (`]`). When this flag is present, UAs must match the attribute's value ASCII case-insensitively (i.e. [a-z] and [A-Z] are considered equivalent).

Alternately, the attribute selector may include the identifier `s` before the closing bracket (`]`); in this case the UA must match the value case-sensitively, with "identical to" semantics [INFRA],

regardless of document language rules.

Like the rest of Selectors syntax, the `i` and `s` identifiers themselves are ASCII case-insensitive.

EXAMPLE 23

The following rule will style the `frame` attribute when it has a value of `hsides`, whether that value is represented as `hsides`, `HSIDES`, `hSides`, etc. even in an XML environment where attribute values are case-sensitive.

```
[frame=hsides i] { border-style: solid none; }
```

EXAMPLE 24

The following rule will style lists with `type="a"` attributes differently than `type="A"` even though HTML defines the `type` attribute to be case-insensitive.

```
[type="a" s] { list-style: lower-alpha; }
[type="A" s] { list-style: upper-alpha; }
```

Note: Some document models normalize case-insensitive attribute values at parse time such that checking if a string is case-sensitive matching is impossible. Case-sensitive matching via `s` flags is only possible in systems that preserve the original case.

§ 6.4. Attribute selectors and namespaces

The attribute name in an attribute selector is given as a CSS qualified name: a namespace prefix that has been previously declared may be prepended to the attribute name separated by the namespace separator "vertical bar" (`|`). In keeping with the Namespaces in the XML recommendation, default namespaces do not apply to attributes, therefore attribute selectors without a namespace component apply only to attributes that have no namespace (equivalent to '|attr'). An asterisk may be used for the namespace prefix indicating that the selector is to match all attribute names without regard to the attribute's namespace.

An attribute selector with an attribute name containing a namespace prefix that has not been previously declared is an invalid selector.

EXAMPLE 25

CSS examples:

```
@namespace foo "http://www.example.com";
[foo|att=val] { color: blue }
[*|att] { color: yellow }
[|att] { color: green }
[att] { color: green }
```

The first rule will match only elements with the attribute `att` in the "http://www.example.com" namespace with the value "val".

The second rule will match only elements with the attribute `att` regardless of the namespace of the attribute (including no namespace).

The last two rules are equivalent and will match only elements with the attribute `att` where the attribute is not in a namespace.

§ 6.5. Default attribute values in DTDs

Attribute selectors represent attribute values in the document tree. How that document tree is constructed is outside the scope of Selectors. In some document formats default attribute values can be defined in a DTD or elsewhere, but these can only be selected by attribute selectors if they appear in the document tree. Selectors should be designed so that they work whether or not the default values are included in the document tree.

For example, a XML UA may, but is *not* required to, read an “external subset” of the DTD, but *is* required to look for default attribute values in the document’s “internal subset”. (See, e.g., [XML10] for definitions of these subsets.) Depending on the UA, a default attribute value defined in the external subset of the DTD might or might not appear in the document tree.

A UA that recognizes an XML namespace may, but is not required to use its knowledge of that namespace to treat default attribute values as if they were present in the document. (For example, an XHTML UA is not required to use its built-in knowledge of the XHTML DTD. See, e.g., [XML-NAMES] for details on namespaces in XML 1.0.)

Note: Typically, implementations choose to ignore external subsets. This corresponds to the behavior of non-validating processors as defined by the XML specification.

EXAMPLE 26

Example:

Consider an element `EXAMPLE` with an attribute `radix` that has a default value of `"decimal"`. The DTD fragment might be

```
<!ATTLIST EXAMPLE radix (decimal,octal) "decimal">
```

If the style sheet contains the rules

```
EXAMPLE[radix=decimal] { /*... default property settings ...*/ }
EXAMPLE[radix=octal]   { /*... other settings...*/ }
```

the first rule might not match elements whose `radix` attribute is set by default, i.e. not set explicitly. To catch all cases, the attribute selector for the default value must be dropped:

```
EXAMPLE                { /*... default property settings ...*/ }
EXAMPLE[radix=octal]   { /*... other settings...*/ }
```

Here, because the selector "EXAMPLE[radix=octal]" is more specific than the type selector alone, the style declarations in the second rule will override those in the first for elements that have a `radix` attribute value of `"octal"`. Care has to be taken that all property declarations that are to apply only to the default case are overridden in the non-default cases' style rules.

§ 6.6. Class selectors

The ***class selector*** is given as a full stop (. U+002E) immediately followed by an identifier. It represents an element belonging to the class identified by the identifier, as defined by the document language. For example, in [HTML5], [SVG11], and [MATHML] membership in a class is given by the `class` attribute: in these languages it is equivalent to the `~=` notation applied to the local `class` attribute (i.e. `[class~=`*`identifier`*`]`).

EXAMPLE 27

CSS examples:

We can assign style information to all elements with `class~="pastoral"` as follows:

```
*.pastoral { color: green }  /* all elements with class~=pastoral */
```

or just

```
.pastoral { color: green }  /* all elements with class~=pastoral */
```

The following assigns style only to H1 elements with `class~="pastoral"`:

```
H1.pastoral { color: green }  /* H1 elements with class~=pastoral */
```

Given these rules, the first `H1` instance below would not have green text, while the second would:

```
<H1>Not green</H1>
<H1 class="pastoral">Very green</H1>
```

The following rule matches any `<P>` element whose `class` attribute has been assigned a list of whitespace-separated values that includes both `pastoral` and `marine`:

```
p.pastoral.marine { color: green }
```

This rule matches when `class="pastoral blue aqua marine"` but does not match for `class="pastoral blue"`.

Note: Because CSS gives considerable power to the "class" attribute, authors could conceivably design their own "document language" based on elements with almost no associated presentation (such as `<div>` and `<span>` in HTML) and assigning style information through the "class" attribute. Authors should avoid this practice since the structural elements of a document language often have recognized and accepted meanings and author-defined classes may not.

Note: If an element has multiple class attributes, their values must be concatenated with spaces between the values before searching for the class. As of this time the working group is not aware of any manner in which this situation can be reached, however, so this behavior is explicitly non-normative in this specification.

When matching against a document which is in quirks mode, class names must be matched ASCII case-insensitively; class selectors are otherwise case-sensitive, only matching class names they are identical to. [INFRA]

§ 6.7. ID selectors

Document languages may contain attributes that are declared to be of type ID. What makes attributes of type ID special is that no two such attributes can have the same value in a conformant document, regardless of the type of the elements that carry them; whatever the document language, an ID typed attribute can be used to uniquely identify its element. In HTML all ID attributes are named `id`; XML applications may name ID attributes differently, but the same restriction applies. Which attribute on an element is considered the “ID attribute“ is defined by the document language.

An ***ID selector*** consists of a “number sign” (U+0023, `#`) immediately followed by the ID value, which must be a CSS identifier. An ID selector represents an element instance that has an identifier that matches the identifier in the ID selector. (It is possible in non-conforming documents for multiple elements to match a single ID selector.)

EXAMPLE 28

Examples: The following ID selector represents an `<h1>` element whose ID-typed attribute has the value "chapter1":

```
h1#chapter1
```

The following ID selector represents any element whose ID-typed attribute has the value "chapter1":

```
#chapter1
```

The following selector represents any element whose ID-typed attribute has the value "z98y".

```
*#z98y
```

Note: In XML 1.0 [XML10], the information about which attribute contains an element’s IDs is contained in a DTD or a schema. When parsing XML, UAs do not always read the DTD, and thus may not know what the ID of an element is (though a UA may have namespace-specific knowledge that allows it to determine which attribute is the ID attribute for that namespace). If a style sheet author knows or suspects that a UA may not know what the ID of an element is, he should use normal attribute selectors instead: "[name=p371]‘ instead of ’#p371".

If an element has multiple ID attributes, all of them must be treated as IDs for that element for the purposes of the ID selector. Such a situation could be reached using mixtures of xml:id, DOM3 Core, XML DTDs, and namespace-specific knowledge.

When matching against a document which is in quirks mode, IDs must be matched ASCII case-insensitively; ID selectors are otherwise case-sensitive, only matching IDs they are identical to. [INFRA]

§ 7. Linguistic Pseudo-classes

§ 7.1. The Directionality Pseudo-class: ‘:dir()’

The ***‘:dir()’*** pseudo-class allows the author to write selectors that represent an element based on its directionality as determined by the document language. For example, [HTML5] defines how to determine the directionality of an element, based on a combination of the `dir` attribute, the surrounding text, and other factors. As another example, the `its:dir` and `dirRule` element of the Internationalization Tag Set [ITS20] are able to define the directionality of an element in [XML10].

The ‘:dir()’ pseudo-class does not select based on stylistic states—for example, the CSS ‘direction’ property does not affect whether it matches.

The pseudo-class ‘:dir(ltr)’ represents an element that has a directionality of left-to-right (`ltr`). The pseudo-class ‘:dir(rtl)’ represents an element that has a directionality of right-to-left (`rtl`). The argument to ‘:dir()’ must be a single identifier, otherwise the selector is invalid. White space is optionally allowed between the identifier and the parentheses. Values other than `ltr` and `rtl` are not invalid, but do not match anything. (If a future markup spec defines other directionalities, then Selectors may be extended to allow corresponding values.)

The difference between ‘:dir(C)’ and "[dir=C]" is that "[dir=C]" only performs a comparison against a given attribute on the element, while the ‘:dir(C)’ pseudo-class uses the UAs knowledge of the document’s semantics to perform the comparison. For example, in HTML, the directionality of an element inherits so that a child without a `dir` attribute will have the same directionality as its closest ancestor with a valid `dir` attribute. As another example, in HTML, an element that matches "[dir=auto]" will match either ‘:dir(ltr)’ or ‘:dir(rtl)’ depending on the resolved directionality of the elements as determined by its contents. [HTML5]

§ 7.2. The Language Pseudo-class: ‘:lang()’

If the document language specifies how the (human) content language of an element is determined, it is possible to write selectors that represent an element based on its content language. The ***‘:lang()’*** pseudo-class, which accepts a comma-separated list of one or more language ranges, represents an element whose content language is one of the languages listed in its argument. Each ***language range*** in ‘:lang()’ must be a valid CSS <ident> or <string>. (Thus language ranges containing asterisks, for example, must be either correctly escaped or quoted as strings, e.g. ‘:lang(*-Latn)’ or ‘:lang("*-Latn")’.)

Note: The content language of an element is defined by the document language. For example, in HTML [HTML5], the content language is determined by a combination of the `lang` attribute, information from `<meta>` elements, and possibly also the protocol (e.g. from HTTP headers). XML languages can use the `xml:lang` attribute to indicate language information for an element. [XML10]

An element's content language matches a language range if, when represented in BCP 47 syntax [BCP47], it matches that language range in an *extended filtering* operation per [RFC4647] *Matching of Language Tags* (section 3.3.2). For this purpose, a wildcard language range (`"*"`) does not match elements whose language is not tagged (e.g. `lang=""`), but does match elements whose language is tagged as undetermined (`lang=und`). The matching is performed ASCII case-insensitively. The language range does not need to be a valid language code to perform this comparison.

A language range consisting of an empty string (':lang("")') matches (only) elements whose language is not tagged.

Note: It is recommended that documents and protocols indicate language using codes from [BCP47] or its successor, and in the case of XML-based formats, by means of `xml:lang` attributes. [XML10] See "FAQ: Two-letter or three-letter language codes."

EXAMPLE 29

Examples: The two following selectors represent an HTML document that is in Belgian French or German. The two next selectors represent `<q>` quotations in an arbitrary element in Belgian French or German.

```
html:lang(fr-be)
html:lang(de)
:lang(fr-be) > q
:lang(de) > q
```

Note: One difference between ':lang(C)' and the "|=" operator is that the "|=" operator only performs a comparison against a given attribute on the element, while the ':lang(C)' pseudo-class uses the UAs knowledge of the document's semantics to perform the comparison.

EXAMPLE 30

In this HTML example, only the BODY matches "[lang|=fr]" (because it has a LANG attribute) but both the BODY and the P match ‘:lang(fr)’ (because both are in French). The P does not match the "[lang|=fr]" because it does not have a LANG attribute.

```
<body lang=fr>
  <p>Je suis français.</p>
</body>
```

EXAMPLE 31

Another difference between ‘:lang(C)’ and the "|=" operator is that ‘:lang(C)’ performs implicit wildcard matching.

For example, ‘:lang(de-DE)’ will match all of ‘de-DE’, ‘de-DE-1996’, ‘de-Latn-DE’, ‘de-Latf-DE’, ‘de-Latn-DE-1996’, whereas of those "[lang|=de-DE]‘ will only match ’de-DE" and ‘de-DE-1996’.

To perform wildcard matching on the first subtag (the primary language), an asterisk must be used: ‘*-CH’ will match all of ‘de-CH’, ‘it-CH’, ‘fr-CH’, and ‘rm-CH’.

To select against an element’s lang attribute value using this type of language range match, use both the attribute selector and language pseudo-class together, e.g. ‘[lang]:lang(de-DE)’.

Note: Wildcard language matching and comma-separated lists are new in Level 4.

§ 8. Location Pseudo-classes

§ 8.1. The Hyperlink Pseudo-class: ‘:any-link’

The ***‘:any-link’*** pseudo-class represents an element that acts as the source anchor of a hyperlink. For example, in [HTML5], any `<a>` or `<area>` elements with an `href` attribute are hyperlinks, and thus match `:any-link`. It matches an element if the element would match either ‘:link’ or ‘:visited’, and is equivalent to ‘:is(:link, :visited)’.

§ 8.2. The Link History Pseudo-classes: ‘:link’ and ‘:visited’

User agents commonly display unvisited hyperlinks differently from previously visited ones. Selectors provides the pseudo-classes ***‘:link’*** and ***‘:visited’*** to distinguish them:

- The ‘:link’ pseudo-class applies to links that have not yet been visited.
- The ‘:visited’ pseudo-class applies once the link has been visited by the user.

After some amount of time, user agents may choose to return a visited link to the (unvisited) ‘:link’ state.

The two states are mutually exclusive.

> EXAMPLE 32
> The following selector represents links carrying class `footnote` and already visited:
>
> ```
> .footnote:visited
> ```

Since it is possible for style sheet authors to abuse the :link and :visited pseudo-classes to determine which sites a user has visited without the user’s consent, UAs may treat all links as unvisited links or implement other measures to preserve the user’s privacy while rendering visited and unvisited links differently.

§ 8.3. The Local Link Pseudo-class: ‘:local-link’

The ***‘:local-link’*** pseudo-class allows authors to style hyperlinks based on the users current location within a site. It represents an element that is the source anchor of a hyperlink whose target’s absolute URL matches the element’s own document URL. If the hyperlink’s target includes a fragment URL, then the fragment URL of the current URL must also match; if it does not, then the fragment URL portion of the current URL is not taken into account in the comparison.

> EXAMPLE 33
> For example, the following rule prevents links targeting the current page from being underlined when they are part of the navigation list:
>
> ```
> nav :local-link { text-decoration: none; }
> ```

> Note: The current URL of a page can change as a result of user actions such as activating a link targeting a different fragment within the same page; or by use of the `pushState` API; as well as by the more obvious actions of navigating to a different page or following a redirect (which could be initiated by protocols such as HTTP, markup instructions such as `<meta http-equiv="...">`, or scripting instructions). UAs must ensure that ‘:local-link’, as well as the ‘:target’ and ‘:target-within’ pseudo-classes below, respond correctly to all such changes in state.

§ 8.4. The Target Pseudo-class: ':target'

In some document languages, the document's URL can further point to specific elements *within* the document via the URL's fragment. The elements pointed to in this way are the target elements of the document.

> EXAMPLE 34
> In HTML the fragment points to the element in the page with the same ID. The url `https://example.com/index.html#section2`, for example, points to the element with `id="section2"` in the document at `https://example.com/index.html`.

The ***':target'*** pseudo-class matches the document's target elements. If the document's URL has no fragment identifier, then the document has no target elements.

> EXAMPLE 35
> Example:
>
> ```
> p.note:target
> ```
>
> This selector represents a `<p>` element of class `note` that is the target element of the referring URL.

> EXAMPLE 36
> CSS example: Here, the ':target' pseudo-class is used to make the target element red and place an image before it, if there is one:
>
> ```
> :target { color : red }
> :target::before { content : url(target.png) }
> ```

§ 8.5. The Target Container Pseudo-class: ':target-within'

The ***':target-within'*** pseudo-class applies to any element to which the ':target' pseudo class applies as well as to any element whose descendant in the flat tree (including non-element nodes, such as text nodes) matches the conditions for matching ':target'.

§ 8.6. The Reference Element Pseudo-class: ':scope'

In some contexts, selectors are matched with respect to one or more scoping roots, such as when calling the `querySelector()` method in [DOM]. The ***':scope'*** pseudo-class represents this scoping root, and may be either a true element or a virtual one (such as a `DocumentFragment`).

If there is no scoping root then ':scope' represents the root of the document (equivalent to ':root'). Specifications intending for this pseudo-class to match specific elements rather than the document's root element must define their scoping root(s).

A virtual scoping root is some object representing the root of a document fragment, and can be used in selector patterns to represent other elements' relationships to this scoping root, acting as the parent of any root elements in the document fragment it represents. A virtual scoping root is featureless and cannot be the subject of the selector.

> EXAMPLE 37
> For example, if you have a `DocumentFragment` `df`, then `df.querySelectorAll(":scope > .foo")` matches all the '.foo' elements that are "top-level" in the document fragment (those that have the document fragment as their `parentNode`).
>
> However, `df.querySelector(":scope")` will not match anything, as the document fragment itself can't be the subject of the selector.

§ 9. User Action Pseudo-classes

Interactive user interfaces sometimes change the rendering in response to user actions. Selectors provides several ***user action pseudo-classes*** for the selection of an element the user is acting on. (In non-interactive user agents, these pseudo-classes are valid, but never match any element.)

These pseudo-classes are not mutually exclusive. An element can match several such pseudo-classes at the same time.

> EXAMPLE 38
> Examples:
>
> ```
> a:link    /* unvisited links */
> a:visited /* visited links */
> a:hover   /* user hovers */
> a:active  /* active links */
> ```
>
> An example of combining dynamic pseudo-classes:
>
> ```
> a:focus
> a:focus:hover
> ```
>
> The last selector matches `<a>` elements that are in the pseudo-class :focus and in the pseudo-class :hover.

> Note: The specifics of hit-testing, necessary to know when several of the pseudo-classes defined in this section apply, are not yet defined, but will be in the future.

§ 9.1. The Pointer Hover Pseudo-class: ‘:hover’

The ***‘:hover’*** pseudo-class applies while the user designates an element with a pointing device, but does not necessarily activate it. For example, a visual user agent could apply this pseudo-class when the cursor (mouse pointer) hovers over a box generated by the element. Interactive user agents that cannot detect hovering due to hardware limitations (e.g., a pen device that does not detect hovering) are still conforming; the selector will simply never match in such a UA.

An element also matches ‘:hover’ if one of its descendants in the flat tree (including non-element nodes, such as text nodes) matches the above conditions.

Document languages may define additional ways in which an element can match ‘:hover’. For example, [HTML5] defines a labeled control element as matching `:hover` when its `<label>` is hovered.

> Note: Since the ‘:hover’ state can apply to an element because its child is designated by a pointing device, it is possible for ‘:hover’ to apply to an element that is not underneath the pointing device.

The ‘:hover’ pseudo-class can apply to any pseudo-element.

§ 9.2. The Activation Pseudo-class: ‘:active’

The ***‘:active’*** pseudo-class applies while an element is being activated by the user. For example, between the times the user presses the mouse button and releases it. On systems with more than one mouse button, ‘:active’ applies only to the primary or primary activation button (typically the "left" mouse button), and any aliases thereof.

There may be document-language or implementation-specific limits on which elements can become ‘:active’. For example, [HTML5] defines a list of activatable elements.

An element also matches ‘:active’ if one of its descendants in the flat tree (including non-element nodes, such as text nodes) matches the above conditions.

Document languages may define additional ways in which an element can match ‘:active’.

> Note: An element can be both ‘:visited’ and ‘:active’ (or ‘:link’ and ‘:active’).

§ 9.3. The Input Focus Pseudo-class: ‘:focus’

The ***‘:focus’*** pseudo-class applies while an element has the focus (accepts keyboard or mouse events, or other forms of input).

There may be document language or implementation specific limits on which elements can acquire ‘:focus’. For example, [HTML] defines a list of focusable areas.

Document languages may define additional ways in which an element can match ‘:focus’, except that the ‘:focus’ pseudo class must not automatically propagate to the parent element—see ‘:focus-within’ if matching on the parent is desired. (It may still apply to the parent element if made to propagate due to other mechanisms, but not merely due to being the parent.)

> ISSUE 5 There’s a desire from authors to propagate ‘:focus’ from a form control to its associated `<label>` element; the main objection seems to be implementation difficulty. See CSSWG issue (CSS) and WHATWG issue (HTML).

§ 9.4. The Focus-Indicated Pseudo-class: ‘:focus-visible’

While the ‘:focus’ pseudo-class always matches the currently-focused element, UAs only sometimes visibly ***indicate focus*** (such as by drawing a “focus ring”), instead using a variety of heuristics to visibly indicate the focus only when it would be most helpful to the user. The ***‘:focus-visible’*** pseudo-class matches a focused element in these situations only, allowing authors to change the appearance of the focus indicator without changing *when* a focus indicator appears.

EXAMPLE 39

In this example, all focusable elements get a strong yellow outline on ‘:focus-visible’, and links get both a yellow outline and a yellow background on ‘:focus-visible’. These styles are consistent throughout the page and are easily visible due to their bold styling, but do not appear unless the user is likely to need to understand where page focus is.

```
:root {
  --focus-gold: #ffbf47;
}

:focus-visible  {
  outline: 3px solid var(--focus-gold);
}

a:focus-visible {
  background-color: var(--focus-gold);
}
```

EXAMPLE 40

User agents can choose their own heuristics for when to indicate focus; however, the following (non-normative) suggestions can be used as a starting point for when to indicate focus on the currently focused element:

- If the user has expressed a preference (such as via a system preference or a browser setting) to always see a visible focus indicator, indicate focus regardless of any other factors. (Another option may be for the user agent to show its own focus indicator regardless of author styles.)
- If the element which supports keyboard input (such as an `<input>` element, or any other element that would triggers a virtual keyboard to be shown on focus if a physical keyboard were not present), indicate focus.
- If the user interacts with the page via keyboard or some other non-pointing device, indicate focus. (This means keyboard usage may change whether this pseudo-class matches even if it doesn’t affect ‘:focus’).
- If the user interacts with the page via a pointing device (mouse, touchscreen, etc.) and the focused element does not support keyboard input, don’t indicate focus.
- If the previously-focused element indicated focus, and a script causes focus to move elsewhere, the newly focused element should indicate focus.

  Conversely, if the previously-focused element did not indicate focus, and a script causes focus to move elsewhere, the newly focused element should also not indicate focus.

User agents should also use ‘:focus-visible’ to specify the default focus style, so that authors using ‘:focus-visible’ will not also need to disable the default ‘:focus’ style.

§ 9.5. The Focus Container Pseudo-class: ‘:focus-within’

The ***‘:focus-within’*** pseudo-class applies to any element for which the ‘:focus’ pseudo class applies as well as to an element whose descendant in the flat tree (including non-element nodes, such as text nodes) matches the conditions for matching ‘:focus’.

§ 10. Time-dimensional Pseudo-classes

These pseudo-classes classify elements with respect to the currently-displayed or active position in some timeline, such as during speech rendering of a document, or during the display of a video using WebVTT to render subtitles.

CSS does not define this timeline; the host language must do so. If there is no timeline defined for an element, these pseudo-classes must not match the element.

> Note: Ancestors of a ‘:current’ element are also ‘:current’, but ancestors of a ‘:past’ or ‘:future’ element are not necessarily ‘:past’ or ‘:future’ as well. A given element matches at most one of ‘:current’, ‘:past’, or ‘:future’.

§ 10.1. The Current-element Pseudo-class: ‘:current’

The ***‘:current’*** pseudo-class represents the element, or an ancestor of the element, that is currently being displayed.

Its alternate form ***‘:current()’***, like ‘:is()’, takes a list of compound selectors as its argument: it represents the ‘:current’ element that matches the argument or, if that does not match, the innermost ancestor of the ‘:current’ element that does. (If neither the ‘:current’ element nor its ancestors match the argument, then the selector does not represent anything.)

> EXAMPLE 41
> For example, the following rule will highlight whichever paragraph or list item is being read aloud in a speech rendering of the document:
>
> ```
> :current(p, li, dt, dd) {
>   background: yellow;
> }
> ```

§ 10.2. The Past-element Pseudo-class: ':past'

The ***':past'*** pseudo-class represents any element that is defined to occur entirely prior to a ':current' element. For example, the WebVTT spec defines the ':past' pseudo-class relative to the current playback position of a media element. If a time-based order of elements is not defined by the document language, then this represents any element that is a (possibly indirect) previous sibling of a ':current' element.

§ 10.3. The Future-element Pseudo-class: ':future'

The ***':future'*** pseudo-class represents any element that is defined to occur entirely after a ':current' element. For example, the WebVTT spec defines the ':future' pseudo-class relative to the current playback position of a media element. If a time-based order of elements is not defined by the document language, then this represents any element that is a (possibly indirect) next sibling of a ':current' element.

§ 11. Resource State Pseudo-classes

The pseudo-classes in this section apply to elements that represent loaded resources, particularly images/videos, and allow authors to select them based on some quality of their state.

§ 11.1. Media Playback State: the ':playing', ':paused', and ':seeking' pseudo-classes

The ***':playing'*** pseudo-class represents an element that is capable of being "played" or "paused", when that element is "playing". (This includes both when the element is explicitly playing, and when it's temporarily stopped for some reason not connected to user intent, but will automatically resume when that reason is resolved, such as a "buffering" or "stalled" state.)

The ***':paused'*** pseudo-class represents an element that is capable of being "played" or "paused", when that element is "paused" (i.e. *not* "playing"). (This includes both an explicit "paused" state, and other non-playing states like "loaded, hasn't been activated yet", etc.)

The ***':seeking'*** pseudo-class represents an element that is capable of "seeking" when that element is "seeking". (For the `<audio>` and `<video>` elements of HTML, see *HTML* § 4.8.11.9 Seeking.)

§ 11.2. Media Loading State: the ':buffering' and ':stalled' pseudo-classes

The ***':buffering'*** pseudo-class represents an element that is capable of being "played" or "paused",

when that element cannot continue playing because it is actively attempting to obtain media data but has not yet obtained enough data to resume playback. (Note that the element is still considered to be "playing" when it is "buffering". Whenever ':buffering' matches an element, ':playing' also matches the element.)

The ***':stalled'*** pseudo-class represents an element when that element cannot continue playing because it is actively attempting to obtain media data but it has failed to receive any data for some amount of time. For the `<audio>` and `<video>` elements of HTML, this amount of time is the media element stall timeout. [HTML] (Note that, like with the ':buffering' pseudo-class, the element is still considered to be "playing" when it is "stalled". Whenever ':stalled' matches an element, ':playing' also matches the element.)

§ 11.3. Sound State: the ':muted' and ':volume-locked' pseudo-classes

The ***':muted'*** pseudo-class represents an element capable of making sound when that element is "muted" (forced silent). (For the `<audio>` and `<video>` elements of HTML, see muted. [HTML])

The ***':volume-locked'*** pseudo-class represents an element capable of making sound when programmatically changing the element's volume does not change the element's effective media volume.

§ 12. Element Display State Pseudo-classes

§ 12.1. Collapse State: the ':open' and ':closed' pseudo-class

The ***':open'*** pseudo-class represents an element that has both "open" and "closed" states, and which is currently in the "open" state.

The ***':closed'*** pseudo-class represents an element that has both "open" and "closed" states, and which is currently in the closed state.

Exactly what "open" and "closed" mean is host-language specific, but exemplified by elements such as HTML's `<details>`, `<select>`, and `<dialog>` elements, all of which can be toggled "open" to display more content (or any content at all, in the case of `<dialog>`).

> Note: Being "open" or "closed" is a semantic state. An element not currently being displayed (for example, one that has 'visibility: collapse', or belongs to a 'display: none' subtree) can still be "open" and will match ':open'.

§ 12.2. Modal (Exclusive Interaction) State: the ':modal' pseudo-class

The ***':modal'*** pseudo-class represents an element which is in a state that excludes all interaction with elements outside it until it has been dismissed. Multiple elements can be ':modal' simultaneously, with only one of them active (able to receive input).

> EXAMPLE 42
> For example, the `<dialog>` element is ':modal' when opened with the `showModal()` API. Similarly, a ':fullscreen' element is also ':modal' when opened with the `requestFullscreen()` API, since this prevents interaction with the rest of the page.

§ 12.3. Fullscreen Presentation State: the ':fullscreen' pseudo-class

The ***':fullscreen'*** pseudo-class represents an element which is displayed in a mode that takes up most (usually all) of the screen, such as that defined by the Fullscreen API. [FULLSCREEN]

§ 12.4. Picture-in-Picture Presentation State: the ':picture-in-picture' pseudo-class

The ***':picture-in-picture'*** pseudo-class represents an element which is displayed in a mode that takes up most (usually all) of the viewport, and whose viewport is confined to part of the screen while being displayed over other content, for example when using the Picture-in-Picture API. [picture-in-picture]

§ 13. The Input Pseudo-classes

The pseudo-classes in this section mostly apply to elements that take user input, such as HTML's `<input>` element.

§ 13.1. Input Control States

§ 13.1.1. The ':enabled' and ':disabled' Pseudo-classes

The ***':enabled'*** pseudo-class represents user interface elements that are in an enabled state; such elements must have a corresponding disabled state.

Conversely, the ***':disabled'*** pseudo-class represents user interface elements that are in a disabled state; such elements must have a corresponding enabled state.

What constitutes an enabled state, a disabled state, and a user interface element is host-language-dependent. In a typical document most elements will be neither ':enabled' nor ':disabled'. For example, [HTML5] defines non-disabled interactive elements to be ':enabled', and any such

elements that are explicitly disabled to be ‘:disabled’.

> Note: CSS properties that might affect a user’s ability to interact with a given user interface element do not affect whether it matches ‘:enabled’ or ‘:disabled’; e.g., the ‘display’ and ‘visibility’ properties have no effect on the enabled/disabled state of an element.

§ 13.1.2. The Mutability Pseudo-classes: ‘:read-only’ and ‘:read-write’

An element matches ***‘:read-write’*** if it is user-alterable, as defined by the document language. Otherwise, it is ***‘:read-only’***.

For example, in [HTML5] a non-disabled non-readonly `<input>` element is ‘:read-write’, as is any element with the `contenteditable` attribute set to the true state.

§ 13.1.3. The Placeholder-shown Pseudo-class: ‘:placeholder-shown’

Input elements can sometimes show placeholder text as a hint to the user on what to type in. See, for example, the `placeholder` attribute in [HTML5]. The ***‘:placeholder-shown’*** pseudo-class matches an input element that is showing such placeholder text, whether that text is given by an attribute or a real element, or is otherwise implied by the UA.

> EXAMPLE 43
> For example, according to the semantics of [HTML] the `placeholder` attribute on the `<input>` element provide placeholder text, as does the first `<option>` element of a `<select>` under certain conditions. The ‘:placeholder-shown’ class thus applies whenever such placeholder text is shown.

§ 13.1.4. The Automatic Input Pseudo-class: ‘:autofill’

The ***‘:autofill’*** pseudo-class represents input elements that have been automatically filled by the user agent, and have not been subsequently altered by the user.

§ 13.1.5. The Default-option Pseudo-class: ‘:default’

The ***‘:default’*** pseudo-class applies to the one or more UI elements that are the default among a set of similar elements. Typically applies to context menu items, buttons and select lists/menus.

One example is the default submit button among a set of buttons. Another example is the default option from a popup menu. In a select-many group (such as for pizza toppings), multiple elements

can match ‘:default’. For example, [HTML5] defines that ‘:default’ matches the “default button” in a form, the initially-selected `<option>`(s) in a `<select>`, and a few other elements.

§ 13.2. Input Value States

§ 13.2.1. The Selected-option Pseudo-class: ‘:checked’

Radio and checkbox elements can be toggled by the user. Some menu items are “checked” when the user selects them. When such elements are toggled “on” the ***‘:checked’*** pseudo-class applies. For example, [HTML5] defines that checked checkboxes, radio buttons, and selected `<option>` elements match ‘:checked’.

While the ‘:checked’ pseudo-class is dynamic in nature, and can altered by user action, since it can also be based on the presence of semantic attributes in the document (such as the `selected` and `checked` attributes in [HTML5]), it applies to all media.

> EXAMPLE 44
> An unchecked checkbox can be selected by using the negation pseudo-class:
>
> ```
> input[type=checkbox]:not(:checked)
> ```

§ 13.2.2. The Indeterminate-value Pseudo-class: ‘:indeterminate’

The ***‘:indeterminate’*** pseudo-class applies to UI elements whose value is in an indeterminate state. For example, radio and checkbox elements can be toggled between checked and unchecked states, but are sometimes in an indeterminate state, neither checked nor unchecked. Similarly a progress meter can be in an indeterminate state when the percent completion is unknown. For example, [HTML5] defines how checkboxes can be made to match ‘:indeterminate’.

Like the ‘:checked’ pseudo-class, ‘:indeterminate’ applies to all media. Components of a radio-group initialized with no pre-selected choice, for example, would be ‘:indeterminate’ even in a static display.

§ 13.3. Input Value-checking

§ 13.3.1. The Empty-Value Pseudo-class: ‘:blank’

The ***‘:blank’*** pseudo-class applies to user-input elements whose input value is empty (consists of the empty string or otherwise null input).

EXAMPLE 45

Examples of ':blank' user-input elements would be a `<textarea>` element whose contents are empty, or an `<input>` field whose value is empty. Note that the value under consideration here is the value that would be submitted (see A form control's value in [HTML]), which in HTML does not necessarily correspond to the value of the element's `value` attribute.

Note: This selector is at-risk.

§ 13.3.2. The Validity Pseudo-classes: ':valid' and ':invalid'

An element is ***':valid'*** or ***':invalid'*** when its contents or value is, respectively, valid or invalid with respect to data validity semantics defined by the document language (e.g. [XFORMS11] or [HTML5]). An element which lacks data validity semantics is neither ':valid' nor ':invalid'.

Note: There is a difference between an element which has no constraints, and thus would always be ':valid', and one which has no data validity semantics at all, and thus is neither ':valid' nor ':invalid'. In HTML, for example, an `<input type="text">` element may have no constraints, but a `<p>` element has no validity semantics at all, and so it never matches either of these pseudo-classes.

§ 13.3.3. The Range Pseudo-classes: ':in-range' and ':out-of-range'

The ***':in-range'*** and ***':out-of-range'*** pseudo-classes apply only to elements that have range limitations. An element is ':in-range' or ':out-of-range' when the value that the element is bound to is in range or out of range with respect to its range limits as defined by the document language. An element that lacks data range limits or is not a form control is neither ':in-range' nor ':out-of-range'. E.g. a slider element with a value of 11 presented as a slider control that only represents the values from 1-10 is :out-of-range. Another example is a menu element with a value of "E" that happens to be presented in a popup menu that only has choices "A", "B" and "C".

§ 13.3.4. The Optionality Pseudo-classes: ':required' and ':optional'

A form element is ***':required'*** or ***':optional'*** if a value for it is, respectively, required or optional before the form it belongs to can be validly submitted. Elements that are not form elements are neither required nor optional.

§ 13.3.5. The User-interaction Pseudo-classes: ':user-valid' and ':user-invalid'

The ***':user-invalid'*** and the ***':user-valid'*** pseudo-classes represent an element with incorrect or correct input, respectively, but only *after* the user has significantly interacted with it.

The ':user-invalid' pseudo-class must match an ':invalid', ':out-of-range', or blank-but-':required' elements between the time the user has attempted to submit the form and before the user has interacted again with the form element.

The ':user-valid' pseudo-class must match a ':valid' element between the time the user has attempted to submit the form and before the user has interacted again with the form element.

User-agents may allow them to match such elements at other times, as would be appropriate for highlighting an error to the user. For example, a UA may choose to have ':user-invalid' match an ':invalid' element once the user has typed some text into it and changed the focus to another element, and to stop matching only after the user has successfully corrected the input.

EXAMPLE 46

For example, the input in the following document fragment would match ':invalid' as soon as the page is loaded (because it the initial value violates the max-constraint), but it won't match ':user-invalid' until the user significantly interacts with the element, or attempts to submit the form it's part of.

```
<form>
  <label>
    Volume:
    <input name='vol' type=number min=0 max=10 value=11>
  </label>
  ...
</form>
```

ISSUE 6 Cross-check with ':-moz-ui-invalid'.

ISSUE 7 Evaluate proposed :dirty pseudo-class

ISSUE 8 Clarify that this (and ':invalid'/':valid') can apply to form and fieldset elements.

§ 14. Tree-Structural pseudo-classes

Selectors introduces the concept of ***structural pseudo-classes*** to permit selection based on extra information that lies in the document tree but cannot be represented by other simple selectors or combinators.

Standalone text and other non-element nodes are not counted when calculating the position of an element in the list of children of its parent. When calculating the position of an element in the list of children of its parent, the index numbering starts at 1.

The structural pseudo-classes only apply to elements in the document tree; they must never match pseudo-elements.

§ 14.1. ':root' pseudo-class

The ***':root'*** pseudo-class represents an element that is the root of the document.

For example, in a DOM document, the ':root' pseudo-class matches the root element of the Document object. In HTML, this would be the `<html>` element (unless scripting has been used to modify the document).

§ 14.2. ':empty' pseudo-class

The ***':empty'*** pseudo-class represents an element that has no children except, optionally, document white space characters. In terms of the document tree, only element nodes and content nodes (such as [DOM] text nodes, and entity references) whose data has a non-zero length must be considered as affecting emptiness; comments, processing instructions, and other nodes must not affect whether an element is considered empty or not.

> EXAMPLE 47
> Examples: 'p:empty' is a valid representation of the `<p>` elements in the following HTML fragment:
>
> ```
> 
> 
>  
> 
> ```
>
> 'div:empty' is not a valid representation of the `<div>` elements in the following fragment:
>
> ```
> text
> 
>  
> bla
> this is not :empty
> ```

> Note: In Level 2 and Level 3 of Selectors, ‘:empty’ did not match elements that contained only white space. This was changed so that that—given white space is largely collapsible in HTML and is therefore used for source code formatting, and especially because elements with omitted end tags are likely to absorb such white space into their DOM text contents—elements which authors perceive of as empty can be selected by this selector, as they expect.

§ 14.3. Child-indexed Pseudo-classes

The pseudo-classes defined in this section select elements based on their index amongst their inclusive siblings.

> Note: Selectors 3 described these selectors as selecting elements based on their index in the child list of their parents. (This description survives in the name of this very section, and the names of several of the pseudo-classes.) As there was no reason to exclude them from matching elements without parents, or with non-element parents, they have been rephrased to refer to an element’s relative index amongst its siblings.

§ 14.3.1. ‘:nth-child()’ pseudo-class

The ***‘:nth-child(*An+B *[of* S*]?)’*** pseudo-class notation represents elements that are among *An*+*B*th elements from the list composed of their inclusive siblings that match the selector list *S*, which is a <complex-selector-list> parsed as a forgiving selector list. If *S* is omitted, it defaults to ‘*|*’.

The *An*+*B* notation and its interpretation are defined in *CSS Syntax 3* § 6 The An+B microsyntax; it represents any index $i = An + B$ for any non-negative integer n.

> Note: For these purposes, the list of elements is **1-indexed**; that is, the first child of an element has index 1, and will be matched by ‘:nth-child(2n+1)’, because when `n=0` the expression evaluates to ‘1’.

For example, this selector could address every other row in a table, and could be used to alternate the color of paragraph text in a cycle of four.

EXAMPLE 48

Examples:

```
:nth-child(even)   /* represents the 2nd, 4th, 6th, etc elements
:nth-child(10n-1)  /* represents the 9th, 19th, 29th, etc elements */
:nth-child(10n+9)  /* Same */
:nth-child(10n+-1) /* Syntactically invalid, and would be ignored */
```

Note: The specificity of the ‘:nth-child()’ pseudo-class is the specificity of a single pseudo-class plus, if *S* is specified, the specificity of the most specific complex selector in *S*. See § 17 Calculating a selector’s specificity. Thus ‘S:nth-child(An+B)’ and ‘:nth-child(An+B of S)’ have the exact same specificity, although they do differ in behavior (see example below).

EXAMPLE 49

By passing a selector argument, we can select the Nth element that matches that selector. For example, the following selector matches the first three “important” list items, denoted by the ‘.important’ class:

```
:nth-child(-n+3 of li.important)
```

Note that this is different from moving the selector outside of the function, like:

```
li.important:nth-child(-n+3)
```

This selector instead just selects the first three children if they also happen to be "important" list items.

EXAMPLE 50

Here's another example of using the selector argument, to ensure that zebra-striping a table works correctly.

Normally, to zebra-stripe a table's rows, an author would use CSS similar to the following:

```
tr {
  background: white;
}
tr:nth-child(even) {
  background: silver;
}
```

However, if some of the rows are hidden and not displayed, this can break up the pattern, causing multiple adjacent rows to have the same background color. Assuming that rows are hidden with the '[hidden]' attribute in HTML, the following CSS would zebra-stripe the table rows robustly, maintaining a proper alternating background regardless of which rows are hidden:

```
tr {
  background: white;
}
tr:nth-child(even of :not([hidden])) {
  background: silver;
}
```

§ 14.3.2. ':nth-last-child()' pseudo-class

The ***':nth-last-child(*An+B *[of* S*]?)'*** pseudo-class notation represents elements that are among *An*+*B*th elements from the list composed of their inclusive siblings that match the selector list *S*, counting backwards from the end. *S* is <complex-selector-list> parsed as a forgiving selector list. If *S* is omitted, it defaults to '*|*'.

Note: The specificity of the ':nth-last-child()' pseudo-class, like the ':nth-child()' pseudo-class, combines the specificity of a regular pseudo-class with that of its selector argument *S*. See § 17 Calculating a selector's specificity.

The CSS Syntax Module [CSS3SYN] defines the *An*+*B* notation.

EXAMPLE 51

Examples:

```
tr:nth-last-child(-n+2)    /* represents the two last rows of an HTML table */

foo:nth-last-child(odd)    /* represents all odd foo elements in their parent
                              counting from the last one */
```

14.3.3. ‘:first-child’ pseudo-class

The ***‘:first-child’*** pseudo-class represents an element that if first among its inclusive siblings. Same as ‘:nth-child(1)’.

EXAMPLE 52

Examples: The following selector represents a `<p>` element that is the first child of a `<div>` element:

```
div > p:first-child
```

This selector can represent the `p` inside the `div` of the following fragment:

```
<p> The last P before the note.</p>
<div class="note">
   <p> The first P inside the note.</p>
</div>
```

but cannot represent the second `p` in the following fragment:

```
<p> The last P before the note.</p>
<div class="note">
   <h2> Note </h2>
   <p> The first P inside the note.</p>
</div>
```

The following two selectors are usually equivalent:

```
* > a:first-child /* a is first child of any element */
a:first-child /* Same (assuming a is not the root element) */
```

14.3.4. ‘:last-child’ pseudo-class

The ***‘:last-child’*** pseudo-class represents an element that is last among its inclusive siblings. Same

as ‘:nth-last-child(1)’.

> EXAMPLE 53
> Example: The following selector represents a list item `li` that is the last child of an ordered list `ol`.
>
> ```
> ol > li:last-child
> ```

14.3.5. ‘:only-child’ pseudo-class

The ***‘:only-child’*** pseudo-class represents an element that has no siblings. Same as ‘:first-child:last-child’ or ‘:nth-child(1):nth-last-child(1)’, but with a lower specificity.

14.4. Typed Child-indexed Pseudo-classes

The pseudo-classes in this section are similar to the Child Index Pseudo-classes, but they resolve based on an element’s index **among elements of the same type (tag name)** in their sibling list.

14.4.1. ‘:nth-of-type()’ pseudo-class

The ***‘:nth-of-type(An+B)’*** pseudo-class notation represents the same elements that would be matched by ‘:nth-child(|An+B| of *S*)’, where *S* is a type selector and namespace prefix matching the element in question. For example, when considering whether an HTML `<img>` element matches this pseudo-class, the *S* in question is ‘html|img’ (assuming an appropriate `html` namespace is declared).

> EXAMPLE 54
> CSS example: This allows an author to alternate the position of floated images:
>
> ```
> img:nth-of-type(2n+1) { float: right; }
> img:nth-of-type(2n) { float: left; }
> ```

> Note: If the type of the element is known ahead of time, this pseudo-class is equivalent to using ‘:nth-child()’ with a type selector. That is, ‘img:nth-of-type(2)’ is equivalent to ‘*:nth-child(2 of img)’.

14.4.2. ‘:nth-last-of-type()’ pseudo-class

The ***':nth-last-of-type(An+B)'*** pseudo-class notation represents the same elements that would be matched by ':nth-last-child(|An+B| of *S*)', where *S* is a type selector and namespace prefix matching the element in question. For example, when considering whether an HTML `<img>` element matches this pseudo-class, the *S* in question is 'html|img' (assuming an appropriate `html` namespace is declared).

> EXAMPLE 55
> Example: To represent all `h2` children of an XHTML `body` except the first and last, one could use the following selector:
>
> ```
> body > h2:nth-of-type(n+2):nth-last-of-type(n+2)
> ```
>
> In this case, one could also use ':not()', although the selector ends up being just as long:
>
> ```
> body > h2:not(:first-of-type):not(:last-of-type)
> ```

14.4.3. ':first-of-type' pseudo-class

The ***':first-of-type'*** pseudo-class represents the same element as ':nth-of-type(1)'.

> EXAMPLE 56
> Example: The following selector represents a definition title `dt` inside a definition list `dl`, this `dt` being the first of its type in the list of children of its parent element.
>
> ```
> dl dt:first-of-type
> ```
>
> It is a valid description for the first two `dt` elements in the following example but not for the third one:
>
> ```
> <dl>
>   <dt>gigogne</dt>
>   <dd>
>     <dl>
>       <dt>fusée</dt>
>       <dd>multistage rocket</dd>
>       <dt>table</dt>
>       <dd>nest of tables</dd>
>     </dl>
>   </dd>
> </dl>
> ```

14.4.4. ':last-of-type' pseudo-class

The ***':last-of-type'*** pseudo-class represents the same element as ':nth-last-of-type(1)'.

> EXAMPLE 57
> Example: The following selector represents the last data cell `td` of a table row `tr`.
>
> ```
> tr > td:last-of-type
> ```

14.4.5. ':only-of-type' pseudo-class

The ***':only-of-type'*** pseudo-class represents the same element as ':first-of-type:last-of-type'.

15. Combinators

15.1. Descendant combinator ( )

At times, authors may want selectors to describe an element that is the descendant of another element in the document tree (e.g., "an `<em>` element that is contained within an `<H1>` element"). The ***descendant combinator*** expresses such a relationship.

A descendant combinator is whitespace that separates two compound selectors.

A selector of the form 'A B' represents an element `B` that is an arbitrary descendant of some ancestor element `A`.

EXAMPLE 58

Examples: For example, consider the following selector:

```
h1 em
```

It represents an `<em>` element being the descendant of an `<h1>` element. It is a correct and valid, but partial, description of the following fragment:

```
<h1>This <span class="myclass">headline
is <em>very</em> important</span></h1>
```

The following selector:

```
div * p
```

represents a `<p>` element that is a grandchild or later descendant of a `<div>` element. Note the whitespace on either side of the "*" is not part of the universal selector; the whitespace is a combinator indicating that the `div` must be the ancestor of some element, and that that element must be an ancestor of the `p`. The following selector, which combines descendant combinators and attribute selectors, represents an element that (1) has the `href` attribute set and (2) is inside a `p` that is itself inside a `div`:

```
div p *[href]
```

§ 15.2. Child combinator (>)

A ***child combinator*** describes a childhood relationship between two elements. A child combinator is made of the "greater-than sign" (U+003E, '>') code point and separates two compound selectors.

EXAMPLE 59

Examples: The following selector represents a `<p>` element that is child of `body`:

```
body > p
```

The following example combines descendant combinators and child combinators.

```
div ol>li p
```

It represents a `<p>` element that is a descendant of an `<li>` element; the li element must be the child of an `<ol>` element; the ol element must be a descendant of a `div`. Notice that the optional white space around the ">" combinator has been left out.

For information on selecting the first child of an element, please see the section on the ':first-child'

pseudo-class above.

§ 15.3. Next-sibling combinator (+)

The ***next-sibling combinator*** is made of the “plus sign” (U+002B, ‘+’) code point that separates two compound selectors. The elements represented by the two compound selectors share the same parent in the document tree and the element represented by the first compound selector immediately precedes the element represented by the second one. Non-element nodes (e.g. text between elements) are ignored when considering the adjacency of elements.

> EXAMPLE 60
> Examples: The following selector represents a `<p>` element immediately following a `<math>` element:
>
> ```
> math + p
> ```
>
> The following selector is conceptually similar to the one in the previous example, except that it adds an attribute selector — it adds a constraint to the `<h1>` element, that it must have `class="opener"`:
>
> ```
> h1.opener + h2
> ```

§ 15.4. Subsequent-sibling combinator (~)

The ***subsequent-sibling combinator*** is made of the "tilde" (U+007E, ‘~’) code point that separates two compound selectors. The elements represented by the two compound selectors share the same parent in the document tree and the element represented by the first compound selector precedes (not necessarily immediately) the element represented by the second one.

> EXAMPLE 61
>
> ```
> h1 ~ pre
> ```
>
> represents a `<pre>` element following an `h1`. It is a correct and valid, but partial, description of:
>
> ```
> <h1>Definition of the function a</h1>
> Function a(x) has to be applied to all figures in the table.
> <pre>function a(x) = 12x/13.5</pre>
> ```

§ 16. Grid-Structural Selectors

The double-association of a cell in a 2D grid (to its row and column) cannot be represented by parentage in a hierarchical markup language. Only one of those associations can be represented hierarchically: the other must be explicitly or implicitly defined in the document language semantics. In both HTML and DocBook, two of the most common hierarchical markup languages, the markup is row-primary (that is, the row associations are represented hierarchically); the columns must be implied. To be able to represent such implied column-based relationships, the column combinator and the ':nth-col()' and ':nth-last-col()' pseudo-classes are defined. In a column-primary format, these pseudo-classes match against row associations instead.

§ 16.1. Column combinator (| |)

The ***column combinator***, which consists of two pipes ('||') represents the relationship of a column element to a cell element belonging to the column it represents. Column membership is determined based on the semantics of the document language only: whether and how the elements are presented is not considered. If a cell element belongs to more than one column, it is represented by a selector indicating membership in any of those columns.

EXAMPLE 62

The following example makes cells C, E, and G gray.

```
col.selected || td {
  background: gray;
  color: white;
  font-weight: bold;
}

<table>
  <col span="2">
  <col class="selected">
  <tr><td>A <td>B <td>C
  <tr><td colspan="2">D <td>E
  <tr><td>F <td colspan="2">G
</table>
```

§ 16.2. ':nth-col()' pseudo-class

The ***':nth-col(An+B)'*** pseudo-class notation represents a cell element belonging to a column that has *An*+*B*-1 columns **before** it, for any positive integer or zero value of `n`. Column membership is determined based on the semantics of the document language only: whether and how the elements are presented is not considered. If a cell element belongs to more than one column, it is represented by a selector indicating any of those columns.

The CSS Syntax Module [CSS3SYN] defines the *An+B* notation.

§ 16.3. ‘:nth-last-col()’ pseudo-class

The ***‘:nth-last-col(An+B)’*** pseudo-class notation represents a cell element belonging to a column that has *An+B*-1 columns **after** it, for any positive integer or zero value of n. Column membership is determined based on the semantics of the document language only: whether and how the elements are presented is not considered. If a cell element belongs to more than one column, it is represented by a selector indicating any of those columns.

The CSS Syntax Module [CSS3SYN] defines the *An+B* notation.

§ 17. Calculating a selector’s specificity

A selector’s ***specificity*** is calculated for a given element as follows:

- count the number of ID selectors in the selector (= *A*)
- count the number of class selectors, attributes selectors, and pseudo-classes in the selector (= *B*)
- count the number of type selectors and pseudo-elements in the selector (= *C*)
- ignore the universal selector

If the selector is a selector list, this number is calculated for each selector in the list. For a given matching process against the list, the specificity in effect is that of the most specific selector in the list that matches.

A few pseudo-classes provide “evaluation contexts” for other selectors, and so have their specificity defined specially:

- The specificity of an ‘:is()’, ‘:not()’, or ‘:has()’ pseudo-class is replaced by the specificity of the most specific complex selector in its selector list argument.
- Analogously, the specificity of an ‘:nth-child()’ or ‘:nth-last-child()’ selector is the specificity of the pseudo class itself (counting as one pseudo-class selector) plus the specificity of the most specific complex selector in its selector list argument (if any).
- The specificity of a ‘:where()’ pseudo-class is replaced by zero.

EXAMPLE 63

For example:

- ‘:is(em, #foo)’ has a specificity of (1,0,0)—like an ID selector (‘#foo’)—when matched against any of `<em>`, `<p id=foo>`, or `<em id=foo>`.
- ‘.qux:where(em, #foo#bar#baz)’ has a specificity of (0,1,0): only the ‘.qux’ outside the ‘:where()’ contributes to selector specificity.
- ‘:nth-child(even of li, .item)’ has a specificity of (0,2,0)—like a class selector (‘.item’) plus a pseudo-class—when matched against any of `<li>`, `<ul class=item>`, or `<li class=item id=foo>`.
- ‘:not(em, strong#foo)’ has a specificity of (1,0,1)—like a tag selector (‘strong’) combined with an ID selector (‘#foo’)—when matched against any element.

Specificities are compared by comparing the three components in order: the specificity with a larger *A* value is more specific; if the two *A* values are tied, then the specificity with a larger *B* value is more specific; if the two *B* values are also tied, then the specificity with a larger *C* value is more specific; if all the values are tied, the two specificities are equal.

Due to storage limitations, implementations may have limitations on the size of *A*, *B*, or *C*. If so, values higher than the limit must be clamped to that limit, and not overflow.

EXAMPLE 64

Examples:

```
*               /* a=0 b=0 c=0 */
LI              /* a=0 b=0 c=1 */
UL LI           /* a=0 b=0 c=2 */
UL OL+LI        /* a=0 b=0 c=3 */
H1 + *[REL=up]  /* a=0 b=1 c=1 */
UL OL LI.red    /* a=0 b=1 c=3 */
LI.red.level    /* a=0 b=2 c=1 */
#x34y           /* a=1 b=0 c=0 */
#s12:not(FOO)   /* a=1 b=0 c=1 */
.foo :is(.bar, #baz)
                /* a=1 b=1 c=0 */
```

Note: Repeated occurrences of the same simple selector are allowed and do increase specificity.

Note: The specificity of the styles specified in an HTML `style` attribute is described in CSS Style Attributes. [CSSSTYLEATTR]

§ 18. Grammar

Selectors are parsed according to the following grammar:

```
<selector-list> = <complex-selector-list>

<complex-selector-list> = <complex-selector>#

<compound-selector-list> = <compound-selector>#

<simple-selector-list> = <simple-selector>#

<relative-selector-list> = <relative-selector>#


<complex-selector> = <compound-selector> [ <combinator>? <compound-selector> ]*

<relative-selector> = <combinator>? <complex-selector>

<compound-selector> = [ <type-selector>? <subclass-selector>*
                        [ <pseudo-element-selector> <pseudo-class-selector>* ]* ]

<simple-selector> = <type-selector> | <subclass-selector>


<combinator> = '>' | '+' | '~' | [ '|' '|' ]

<type-selector> = <wq-name> | <ns-prefix>? '*'

<ns-prefix> = [ <ident-token> | '*' ]? '|'

<wq-name> = <ns-prefix>? <ident-token>

<subclass-selector> = <id-selector> | <class-selector> |
                      <attribute-selector> | <pseudo-class-selector>

<id-selector> = <hash-token>

<class-selector> = '.' <ident-token>

<attribute-selector> = '[' <wq-name> ']' |
                       '[' <wq-name> <attr-matcher> [ <string-token> | <ident-tok

<attr-matcher> = [ '~' | '|' | '^' | '$' | '*' ]? '='

<attr-modifier> = i | s
```

```
<pseudo-class-selector> = ':' <ident-token> |
                          ':' <function-token> <any-value> ')'

<pseudo-element-selector> = ':' <pseudo-class-selector>
```

In interpreting the above grammar, the following rules apply:

- White space is forbidden:
  - Between any of the top-level components of a <compound-selector> (that is, forbidden between the <type-selector> and <subclass-selector>, or between the <subclass-selector> and <pseudo-element-selector>, etc).
  - Between *any* of the components of a <type-selector> or a <class-selector>.
  - Between the ':'s, or between the ':' and <ident-token> or <function-token>, of a <pseudo-element-selector> or a <pseudo-class-selector>.
  - Between *any* of the components of a <wq-name>.
  - Between the components of an <attr-matcher>.
  - Between the components of a <combinator>.

  Whitespace is *required* between two <compound-selector>s if the <combinator> between them is omitted. (This indicates the descendant combinator is being used.)

- The four Level 2 pseudo-elements (‘::before’, ‘::after’, ‘::first-line’, and ‘::first-letter’) may, for legacy reasons, be represented using the <pseudo-class-selector> grammar, with only a single ":" character at their start.
- In <id-selector>, the <hash-token>’s value must be an identifier.

> Note: A selector is also subject to a variety of more specific syntactic constraints, and adherence to the grammar above is necessary *but not sufficient* for the selector to be considered valid. See § 3.9 Invalid Selectors and Error Handling for additional rules for parsing selectors.

> Note: In general, a <pseudo-element-selector> is only valid if placed at the end of the last <compound-selector> in a <complex-selector>. In some circumstances, however, it can be followed by more <pseudo-element-selector>s or <pseudo-class-selector>s; but these are specified on a case-by-case basis. (For example, the user action pseudo-classes are allowed after any pseudo-element, and the tree-abiding pseudo-elements are allowed after the ‘::slotted()’ pseudo-element.)

§ 18.1. <forgiving-selector-list> and <forgiving-relative-selector-list>

For legacy reasons, the general behavior of a selector list is that if any selector in the list fails to parse (because it uses new or UA-specific selector features, for instance), the entire selector list becomes invalid. This can make it hard to write CSS that uses new selectors and still works correctly in older user agents.

The ***'<forgiving-selector-list>'*** production instead parses each selector in the list individually, simply ignoring ones that fail to parse, so the remaining selectors can still be used.

> Note: Style rules still use the normal, unforgiving selector list behavior. <forgiving-selector-list> is used in some functions, like ':is()', which are similarly generic. Although it does have some minor implications on specificity, wrapping a style rule's selector in ':is()' effectively "upgrades" it to become forgiving.

Syntactically, <forgiving-selector-list> is equivalent to `<any-value>?`. It is then parsed as a forgiving selector list to obtain its actual value.

To ***parse as a forgiving selector list*** given an input *input*:

1. Parse a list of <complex-selector>s from *input*, and let *selector list* be the result.
2. Remove all failure items from *selector list*, and all items that are invalid selectors, then return a <selector-list> representing the remaining items in *selector list*. (This might be empty.)

'<forgiving-relative-selector-list>' is identical to <forgiving-selector-list>, except it parses its components as <relative-selector> rather than <complex-selector>.

§ 19. API Hooks

To aid in the writing of specs that use Selectors concepts, this section defines several API hooks that can be invoked by other specifications.

> ISSUE 9 Are these still necessary now that we have more rigorous definitions for match and invalid selector? Nouns are a lot easier to coordinate across specification than predicates, and details like the exact order of elements returned from `querySelector` seem to make more sense being defined in the DOM specification than in Selectors.

§ 19.1. Parse A Selector

This section defines how to ***parse a selector*** from a string *source*. It returns either a complex selector list, or failure.

1. Let *selector* be the result of parsing *source* as a <selector-list>. If this returns failure, it's an

invalid selector; return failure.

2. If *selector* is an invalid selector for any other reason (such as, for example, containing an undeclared namespace prefix), return failure.
3. Otherwise, return *selector*.

§ 19.2. Parse A Relative Selector

This section defines how to ***parse a relative selector*** from a string *source*. It returns either a complex selector list, or failure.

1. Let *selector* be the result of parsing *source* as a <relative-selector-list>. If this returns failure, it's an invalid selector; return failure.
2. If *selector* is an invalid selector for any other reason (such as, for example, containing an undeclared namespace prefix), return failure.
3. Otherwise, return *selector*.

§ 19.3. Match a Selector Against an Element

This section defines how to ***match a selector against an element***.

APIs using this algorithm must provide a *selector* and an *element*.

Callers may optionally provide:

- one or more scoping roots, for resolving the ':scope' pseudo-class against.

This algorithm returns either success or failure.

For each complex selector in the given *selector* (which is taken to be a list of complex selectors), match the complex selector against *element*, as described in the following paragraph. If the matching returns success for any complex selector, then the algorithm return success; otherwise it returns failure.

To ***match a complex selector against an element***, process it compound selector at a time, in right-to-left order. This process is defined recursively as follows:

- If any simple selectors in the rightmost compound selector does not match the element, return failure.
- Otherwise, if there is only one compound selector in the complex selector, return success.
- Otherwise, consider all possible elements that could be related to this element by the rightmost combinator. If the operation of matching the selector consisting of this selector with

the rightmost compound selector and rightmost combinator removed against any one of these elements returns success, then return success. Otherwise, return failure.

§ 19.4. Match a Selector Against a Pseudo-element

This section defines how to ***match a selector against a pseudo-element***.

APIs using this algorithm must provide a *selector* and a *pseudo-element*. They may optionally provide the same things they may optionally provide to the algorithm to match a selector against an element.

This algorithm returns success or failure.

For each complex selector in the given *selector*, if both:

- the rightmost simple selector in the complex selector matches *pseudo-element*, and
- the result of running match a complex selector against an element on the remainder of the complex selector (with just the rightmost simple selector of its rightmost complex selector removed), *pseudo-element*'s corresponding element, and any optional parameters provided to this algorithm returns success,

then return success.

Otherwise (that is, if this doesn't happen for any of the complex selectors in *selector*), return failure.

§ 19.5. Match a Selector Against a Tree

This section defines how to ***match a selector against a tree***.

APIs using this algorithm must provide a selector, and one or more *root elements* indicating the subtrees that will be searched by the selector. All of the *root elements* must share the same root, or else calling this algorithm is invalid.

They may optionally provide:

- One or more scoping roots indicating the selector is scoped.
- A list of pseudo-elements that are allowed to show up in the match list. If not specified, this defaults to allowing all pseudo-elements.

> ISSUE 10 Only the tree-abiding pseudo-elements are really handled in any way remotely like this.

This algorithm returns a (possibly empty) list of elements.

1. Start with a list of *candidate elements*, which are the *root elements* and all of their descendant elements, sorted in shadow-including tree order, unless otherwise specified.
2. If scoping root were provided, then remove from the *candidate elements* any elements that are not descendants of at least one scoping root.
3. Initialize the *selector match list* to empty.
4. For each *element* in the set of *candidate elements*:
    1. If the result of match a selector against an element for *element* and *selector* is success, add *element* to the *selector match list*.
    2. For each possible pseudo-element associated with *element* that is one of the pseudo-elements allowed to show up in the match list, if the result of match a selector against a pseudo-element for the pseudo-element and *selector* is success, add the pseudo-element to the *selector match list*.

> ISSUE 11 The relative position of pseudo-elements in *selector match list* is undefined. There's not yet a context that exposes this information, but we need to decide on something eventually, before something *is* exposed.

§ Appendix A: Guidance on Mapping Source Documents & Data to an Element Tree

This section is informative.

The element tree structure described by the DOM is powerful and useful, but generic enough to model pretty much any language that describes tree-based data (or even graph-based, with a suitable interpretation).

Some languages, like HTML, already have well-defined procedures for producing a DOM object from a resource. If a given language does not, such a procedure must be defined in order for Selectors to apply to documents in that language.

At minimum, the document language must define what maps to the DOM concept of an "element".

The primary one-to-many relationship between nodes—parent/child in tree-based structures, element/neighbors in graph-based structures—should be reflected as the child nodes of an element.

Other features of the element should be mapped to something that serves a similar purpose to the same feature in DOM:

type

If the elements in the document language have some notion of "type" as a basic distinguisher between different groups of elements, it should be reflected as the "type" feature.
If this "type" can be separated into a "basic" name and a "namespace" that groups names into higher-level groups, the latter should be reflected as the "namespace" feature. Otherwise, the element shouldn't have a "namespace" feature, and the entire name should be reflected as the "type" feature.

id

If some aspect of the element functions as a unique identifier across the document, it should be mapped to the "id" feature.

> Note: While HTML only allows an element to have a single ID, this should not be taken as a general restriction. The important quality of an ID is that each ID should be associated with a single element; a single element can validly have multiple IDs.

classes and attributes

Aspects of the element that are useful for identifying the element, but are not generally unique to elements within a document, should be mapped to the "class" or "attribute" features depending on if they're something equivalent to a "label" (a string by itself) or a "property" (a name/value pair)

pseudo-classes and pseudo-elements

If any elements match any pseudo-classes or have any pseudo-elements, that must be explicitly defined.

> ISSUE 12 Some pseudo-classes are *syntactical*, like ':has()' and ':is()', and thus should always work. Need to indicate that somewhere. Probably the structural pseudos always work whenever the child list is ordered.

EXAMPLE 65

For example, JSONSelect is a library that uses selectors to extract information from JSON documents.

- The "elements" of the JSON document are each array, object, boolean, string, number, or null. The array and object elements have their contents as children.
- Each element's type is its JS type name: "array", "object", etc.
- Children of an object have their key as a class.
- Children of an array match the ':first-child', ':nth-child()', etc pseudo-classes.
- The root object matches ':root'.
- It additionally defines ':val()' and ':contains()' pseudo-classes, for matching boolean/number/string elements with a particular value or which contain a particular substring.

This structure is sufficient to allow powerful, compact querying of JSON documents with selectors.

§ Appendix B: Obsolete but Required `-webkit-` Parsing Quirks for Web Compat

This appendix is normative.

Due to legacy Web-compat constraints, user agents expecting to parse Web documents must support the following features:

- ':-webkit-autofill' must be treated as a legacy selector alias of ':autofill'.
- All other pseudo-elements whose names begin with the string "-webkit-" (matched ASCII case-insensitively) and that are not functional notations must be treated as valid at parse time. (That is, '::-webkit-asdf' is valid at parse time, but '::-webkit-jkl()' is not.) If they're not otherwise recognized and supported, they must be treated as matching nothing, and are ***unknown -webkit- pseudo-elements***.

  Unknown -webkit- pseudo-elements must be serialized in ASCII lowercase.

  ▸ **What's this quirk about?**

§ 20. Changes

§ 20.1. Changes since the 7 May 2022 Working Draft

Significant changes since the 7 May 2022 Working Draft:

- Added ‘:open’ and ‘:closed’ pseudo-classes. (Issue 7319)
- Disallowed pseudo-elements from ‘:has()’ unless explicitly allowed by the pseudo-element’s definition. (Issue 7463)
- Disallowed nesting of ‘:has()’. (Issue 7344)
- Defined matching of ‘::lang("")’ and of elements not tagged with a language. (Issue 6915)
- Untangled the concepts of "scoped" and "relative" selectors completely. (Issue 6399)
    - Removed "absolutize a selector" as well, and just defined relative selector matching in terms of the anchoring element.
- Reverted compound selector limitation on ‘:nth-child()’. (Issue 3760)
- Defined ‘:-webkit-autofill’ legacy selector alias. (Issue 7474)

§ 20.2. Changes since the 21 November 2018 Working Draft

Significant changes since the 21 November 2018 Working Draft:

- Removed the Selector profiles, marked ‘:has()’ as optional and at-risk instead. (Issue 3925)
- Added § 3.6.4 Sub-pseudo-elements to define sub-pseudo-elements and related terminology.
- Added ‘:defined’. (Issue 2258)
- Added ‘:modal’. (Issue 6965)
- Added ‘:fullscreen’ and ‘:picture-in-picture’. (Issue 3796)
- Added ‘:seeking’, ‘:buffering’, and ‘:stalled’ media playback state pseudo-classes. (Issue 3821)
- Added ‘:muted’ and ‘:volume-locked’ sound state pseudo-classes. (Issue 3821 and Issue 3933)
- Added ‘:autofill’. (Issue 5775)
- Added ‘:user-valid’. (Discussion)
- Defined ‘:is()’, ‘:where()’, ‘:has()’, ‘:nth-child()’, and ‘:nth-last-child()’ to not be themselves invalidated when containing an invalid selector. (Issue 3264)
- Limited selectors in ‘:nth-child()’ and ‘:nth-last-child()’ to compound selectors for now. (Issue 3760)
- Clarified case-sensitive string matching by referencing string identity as defined in [INFRA].

- Clarified that UA-provided placeholder text still triggers ‘:placeholder-shown’.
- Rewrote ‘:focus-visible’ definition for clarity.
- Switched reminder note in the grammar section to normative text describing the requirement of whitespace between <compound-selector>s when a <combinator> token is missing.

§ 20.3. Changes since the 2 February 2018 Working Draft

Significant changes since the 2 February 2018 Working Draft:

- Named the zero-specificity selector to ‘:where()’. (Issue 2143)
- Renamed ‘:matches()’ to ‘:is()’. (Issue 3258)
- Redefined ‘:empty’ to ignore white-space–only nodes. (Issue 1967)
- Redefined ‘:blank’ to represent empty user input, rather than empty elements. (Issue 1283)
- Changed the specificity of ‘:is()’, ‘:has()’, and ‘:nth-child()’ to not depend on which selector argument matched. (Issue 1027)
- Dropped the ‘:drop()’ pseudo-classes since HTML dropped the related feature. (Issue 2257)
- Added the case-sensitive flag `s` to the attribute selector. (Issue 2101)
- Added further guidance on ‘:focus-visible’.
- Added Appendix B: Obsolete but Required -webkit- Parsing Quirks for Web Compat defining ‘::-webkit-’ pseudo-element parsing quirk. (Issue 3051)
- Rewrote grammar rules about where white space is allowed for clarity. (See § 18 Grammar.)

§ 20.4. Changes since the 2 May 2013 Working Draft

Significant changes since the 2 May 2013 Working Draft include:

- Added the ‘:target-within’, ‘:focus-within’, ‘:focus-visible’, ‘:playing’, and ‘:paused’ pseudo-classes.
- Added a zero-specificity ‘:matches()’-type pseudo-class, with name TBD.
- Replaced subject indicator (‘!’) feature with ‘:has()’.
- Replaced the ‘:nth-match()’ and ‘:nth-last-match()’ selectors with ‘:nth-child(… of selector)’ and ‘:nth-last-child(… of selector)’.
- Changed the ‘:active-drop-target’, ‘:valid-drop-target’, ‘:invalid-drop-target’ with ‘:drop()’.
- Sketched out an empty-or-whitespace-only selector for discussion (See open issue.)

- Renamed ‘:user-error’ to ‘:user-invalid’. (See Discussion)
- Renamed ‘:nth-column()’/‘:nth-last-column()’ to ‘:nth-col()’/‘:nth-last-col()’ to avoid naming confusion with a potential ‘::column’ pseudo-class.
- Changed the non-functional form of the ‘:local-link’ pseudo-class to account for fragment URLs.
- Removed the functional form of the `:local-link()` pseudo-class and reference combinator for lack of interest.
- Rewrote selectors grammar using the CSS Value Definition Syntax.
- Split out relative selectors from scoped selectors, as these are different concepts that can be independently invoked.
- Moved definition of <An+B> microsyntax to CSS Syntax.

  ISSUE 13 Semantic definition should probably move back here.

- Added new sections:
  - § 3.2 Data Model

    ISSUE 14 Need to define tree for XML.

  - § 19 API Hooks
    - Note that earlier versions of this section defined a section on ***evaluating a selector***, but that section is no longer present. Specifications referencing that section should instead reference the algorithm to match a selector against a tree.
- Removed restriction on combinators within ‘:matches()’ and ‘:not()’; see discussion.
- Defined specificity of a selector list. (Why?)
- Required quotes around ‘:lang()’ values involving an asterisk; only language codes which happen to be CSS identifiers can be used unquoted.

Note: The 1 February 2018 draft included an inadvertent commit of unfinished work; 2 February 2018 has reverted this commit (and fixed some links because why not).

§ 20.5. Changes since the 23 August 2012 Working Draft

Significant changes since the 23 August 2012 Working Draft include:

- Added ‘:placeholder-shown’ pseudo-classes.
- Released some restrictions on ‘:matches()’ and ‘:not()’.

- Defined fast and complete Selectors profiles (now called “live” and “snapshot”).
- Improved definition of specificity to better handle ‘:matches()’.
- Updated grammar.
- Cleaned up definition of <An+B> notation.
- Added definition of scope-relative selectors, changed *scope-constrained* to scope-filtered for less confusion with scope-contained.
- The ‘:local-link()’ pseudo-class now ignores trailing slashes.

§ 20.6. Changes since the 29 September 2011 Working Draft

Significant changes since the 29 September 2011 Working Draft include:

- Added language variant handling per RFC 4647.
- Added scoped selectors.
- Added ‘:user-error’ (now called ‘:user-invalid’).
- Added ‘:valid-drop-target’.
- Changed column combinator from double slash to double pipe.

§ 20.7. Changes Since Level 3

Additions since Level 3:

- Extended ‘:not()’ to accept a selector list.
- Added ‘:is()’ and ‘:where()’ and ‘:has()’.
- Added ‘:scope’.
- Added ‘:any-link’ and ‘:local-link’.
- Added time-dimensional pseudo-classes.
- Added ‘:target-within’, ‘:focus-within’, and ‘:focus-visible’.
- Added ‘:dir()’.
- Expanded ‘:lang()’ to accept wildcard matching and lists of language codes.
- Expanded ‘:nth-child()’ to accept a selector list.
- Merged in input selectors from CSS Basic User Interface Module Level 3 and added back ‘:indeterminate’.
- Added ‘:blank’ and ‘:user-invalid’.

- Added grid-structural (column) selectors.
- Added case-insensitive / case-sensitive attribute-value matching flags.

§ 21. Acknowledgements

The CSS working group would like to thank everyone who contributed to the previous Selectors specifications over the years, as those specifications formed the basis for this one. In particular, the working group would like to extend special thanks to the following for their specific contributions to Selectors Level 4: L. David Baron, Andrew Fedoniouk, Daniel Glazman, Ian Hickson, Grey Hodge, Lachlan Hunt, Anne van Kesteren, Jason Cranford Teague, Lea Verou

§ 22. Privacy and Security Considerations

This specification introduces the following privacy and security considerations:

- The ':visited' pseudo-class can expose information about which sites a user has previously visited, if the UA is not careful to screen from scripting any information that would reveal which elements match it.
- The ':autofill' pseudo-class can expose whether a user has interacted with this form before; however the same information can be derived by observing how quickly the form is filled out.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

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

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

> EXAMPLE 66
>
> This is an example of an informative example.

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

> Note, this is an informative note.

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

> **UAs MUST provide an accessible alternative.**

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

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

authoring tool
: A UA that writes a style sheet.

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

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

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

§ Partial implementations

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

§ Implementations of Unstable and Proprietary Features

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

§ Non-experimental implementations

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

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

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

§ Index

§ Terms defined by this specification

structural pseudo-classes, in § 14

<subclass-selector>, in § 18

subject, in § 3.1

subject of a selector, in § 3.1

subject of the selector, in § 3.1

sub-pseudo-element, in § 3.6.4

subsequent-sibling combinator, in § 15.4

:target, in § 8.4

:target-within, in § 8.5

<type-selector>, in § 18

type selector, in § 5.1

ultimate originating element, in § 3.6.4

universal selector, in § 5.2

unknown -webkit- pseudo-elements, in § Unnumbered section

user action pseudo-class, in § 9

:user-invalid, in § 13.3.5

:user-valid, in § 13.3.5

:valid, in § 13.3.2

:visited, in § 8.2

:volume-locked, in § 11.3

:where(), in § 4.4

White space, in § 3.7

<wq-name>, in § 18

§ Terms defined by reference

[css-color-5] defines the following terms:

a

[css-display-3] defines the following terms:

box tree

display

display type

list-item

visibility

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

::after

::before

::first-letter

::first-line

tree-abiding pseudo-element

[css-scoping-1] defines the following terms:

::shadow

::slotted()

:host

:host-context()

flat tree

shadow host

shadow tree

[css-text-3] defines the following terms:

content language

document white space characters

[css-values-4] defines the following terms:

!

#

*

<ident>

<string>

?

identifier

|

[css-writing-modes-3] defines the following terms:

- direction

[CSS3NAMESPACE] defines the following terms:

- css qualified name
- default namespace

[CSS3SYN] defines the following terms:

- <an+b>
- <any-value>
- <function-token>
- <hash-token>
- <ident-token>
- <string-token>
- parse
- parse a list

[DOM] defines the following terms:

- Document
- DocumentFragment
- descendant
- inclusive sibling
- parentNode
- quirks mode
- root
- shadow-including tree order
- tree

[FULLSCREEN] defines the following terms:

- requestFullscreen()

[HTML] defines the following terms:

- a
- area
- audio
- button
- custom element
- details
- dialog
- div
- effective media volume
- element definition
- em
- h1
- href
- html
- img
- input
- label
- li
- media data
- media element stall timeout
- meta
- object
- ol
- option
- p
- placeholder
- pre
- q
- select
- showModal()
- span
- textarea
- value
- video

[INFRA] defines the following terms:

- ascii case-insensitive
- identical to

[mediacapture-streams] defines the following terms:

- muted

[SELECT] defines the following terms:

- *

[URL] defines the following terms:

- fragment

§ References

§ Normative References

[CSS-DISPLAY-3]
Tab Atkins Jr.; Elika Etemad. *CSS Display Module Level 3*. 3 September 2021. CR. URL: https://www.w3.org/TR/css-display-3/

[CSS-PSEUDO-4]
Daniel Glazman; Elika Etemad; Alan Stearns. *CSS Pseudo-Elements Module Level 4*. 31 December 2020. WD. URL: https://www.w3.org/TR/css-pseudo-4/

[CSS-SCOPING-1]
Tab Atkins Jr.; Elika Etemad. *CSS Scoping Module Level 1*. 3 April 2014. WD. URL: https://www.w3.org/TR/css-scoping-1/

[CSS-TEXT-3]
Elika Etemad; Koji Ishii; Florian Rivoal. *CSS Text Module Level 3*. 5 May 2022. CR. URL: https://www.w3.org/TR/css-text-3/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. 16 December 2021. WD. URL: https://www.w3.org/TR/css-values-4/

[CSS-WRITING-MODES-3]
Elika Etemad; Koji Ishii. *CSS Writing Modes Level 3*. 10 December 2019. REC. URL: https://www.w3.org/TR/css-writing-modes-3/

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

[CSS3NAMESPACE]
Elika Etemad. *CSS Namespaces Module Level 3*. 20 March 2014. REC. URL: https://www.w3.org/TR/css-namespaces-3/

[CSS3SYN]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. 24 December 2021. CR. URL: https://www.w3.org/TR/css-syntax-3/

[DOM]

Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

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

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

[MEDIACAPTURE-STREAMS]
Cullen Jennings; et al. *Media Capture and Streams*. 22 September 2022. CR. URL: https://www.w3.org/TR/mediacapture-streams/

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

[SELECT]
Tantek Çelik; et al. *Selectors Level 3*. 6 November 2018. REC. URL: https://www.w3.org/TR/selectors-3/

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

§ Informative References

[BCP47]
A. Phillips, Ed.; M. Davis, Ed.. *Tags for Identifying Languages*. September 2009. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc5646

[CSS-COLOR-5]
Chris Lilley; et al. *CSS Color Module Level 5*. 28 June 2022. WD. URL: https://www.w3.org/TR/css-color-5/

[CSS3UI]
Tantek Çelik; Florian Rivoal. *CSS Basic User Interface Module Level 3 (CSS3 UI)*. 21 June 2018. REC. URL: https://www.w3.org/TR/css-ui-3/

[CSSSTYLEATTR]
Tantek Çelik; Elika Etemad. *CSS Style Attributes*. 7 November 2013. REC. URL: https://www.w3.org/TR/css-style-attr/

[FULLSCREEN]
Philip Jägenstedt. *Fullscreen API Standard*. Living Standard. URL: https://fullscreen.spec.whatwg.org/

[HTML5]
Ian Hickson; et al. *HTML5*. 27 March 2018. REC. URL: https://www.w3.org/TR/html5/

[ITS20]

David Filip; et al. *Internationalization Tag Set (ITS) Version 2.0*. 29 October 2013. REC. URL: https://www.w3.org/TR/its20/

[MATHML]
Patrick D F Ion; Robert R Miner. *Mathematical Markup Language (MathML) 1.01 Specification*. 7 July 1999. REC. URL: https://www.w3.org/TR/REC-MathML/

[PICTURE-IN-PICTURE]
Francois Beaufort. *Picture-in-Picture*. 25 July 2022. WD. URL: https://www.w3.org/TR/picture-in-picture/

[QUIRKS]
Simon Pieters. *Quirks Mode Standard*. Living Standard. URL: https://quirks.spec.whatwg.org/

[RFC4647]
A. Phillips, Ed.; M. Davis, Ed.. *Matching of Language Tags*. September 2006. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc4647

[SVG11]
Erik Dahlström; et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 16 August 2011. REC. URL: https://www.w3.org/TR/SVG11/

[XFORMS11]
John Boyer. *XForms 1.1*. 20 October 2009. REC. URL: https://www.w3.org/TR/xforms11/

[XML-NAMES]
Tim Bray; et al. *Namespaces in XML 1.0 (Third Edition)*. 8 December 2009. REC. URL: https://www.w3.org/TR/xml-names/

[XML10]
Tim Bray; et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 26 November 2008. REC. URL: https://www.w3.org/TR/xml/

§ Issues Index

ISSUE 1 Pseudo-elements aren't handled here, and should be. ↲

ISSUE 2 Clarify that ':not()' and ':is()' can be used when containing above-mentioned pseudos. ↲

ISSUE 3 Does '::first-line:not(:focus)' match anything? ↲

ISSUE 4 Add comma-separated syntax for multiple-value matching? e.g. [rel ~= next, prev, up, first, last] ↲

ISSUE 5 There's a desire from authors to propagate ':focus' from a form control to its associated `<label>` element; the main objection seems to be implementation difficulty. See CSSWG issue (CSS) and WHATWG issue (HTML).

ISSUE 6 Cross-check with ':-moz-ui-invalid'.

ISSUE 7 Evaluate proposed :dirty pseudo-class

ISSUE 8 Clarify that this (and ':invalid'/':valid') can apply to form and fieldset elements.

ISSUE 9 Are these still necessary now that we have more rigorous definitions for match and invalid selector? Nouns are a lot easier to coordinate across specification than predicates, and details like the exact order of elements returned from `querySelector` seem to make more sense being defined in the DOM specification than in Selectors.

ISSUE 10 Only the tree-abiding pseudo-elements are really handled in any way remotely like this.

ISSUE 11 The relative position of pseudo-elements in *selector match list* is undefined. There's not yet a context that exposes this information, but we need to decide on something eventually, before something *is* exposed.

ISSUE 12 Some pseudo-classes are *syntactical*, like ':has()' and ':is()', and thus should always work. Need to indicate that somewhere. Probably the structural pseudos always work whenever the child list is ordered.

ISSUE 13 Semantic definition should probably move back here.

ISSUE 14 Need to define tree for XML.