

Package ‘selectr’

September 27, 2026

Type Package

Title Translate CSS Selectors to XPath Expressions

Version 0.8-0

License BSD_3_clause + file LICENCE

Encoding UTF-8

Depends R (>= 3.6)

Suggests testthat (>= 3.0.0), XML, xml2

Config/testthat/edition 3

URL <https://sjp.co.nz/projects/selectr/>

BugReports <https://github.com/sjp/selectr/issues>

Language en-GB

Description Translates a CSS selector into an equivalent XPath expression. This allows us to use CSS selectors when working with the 'XML' and 'xml2' packages, which can only evaluate XPath expressions. Also provided are convenience functions for querying XML and HTML documents with CSS selectors. This package was originally a port of the Python package 'cssselect' (<https://cssselect.readthedocs.io/>).

NeedsCompilation no

Author Simon Potter [aut, trl, cre],
Simon Sapin [aut],
Ian Bicking [aut]

Maintainer Simon Potter <simon@sjp.co.nz>

Repository CRAN

Date/Publication 2026-09-27 06:50:02 UTC

Contents

css_to_xpath	2
querySelectorAll	5
selectors	10

Index

15

css_to_xpath	<i>Translate a CSS selector to an equivalent XPath expression.</i>
--------------	--

Description

This function aims to create an XPath expression equivalent to what would be matched by the given CSS selector. The reason the translation is required is because the XML and xml2 packages, being a libxml2 wrappers, can only evaluate XPath expressions.

Using this function, it is possible to search an XML tree without the prerequisite of knowing XPath.

Usage

```
css_to_xpath(selector,
             prefix = "descendant-or-self::",
             translator = "generic")
```

Arguments

selector	A character vector of CSS selectors.
prefix	The prefixes to apply to the resulting XPath expressions. The default or "" are most commonly used.
translator	The type of translator that will be used. Possible options are generic (the default), or html or xhtml.

Details

See [selectors](#) for a table of every combinator, attribute operator and pseudo-class this function supports, along with an example translation for each; the rest of this section explains the reasoning behind the more surprising entries in that table.

Each selector given to this function will be translated to an equivalent XPath expression. Each of selector, prefix and translator must either have length 1, in which case its single value is used for every translation, or the common length of the longer arguments. Unlike base R, a length that is merely a multiple (or a fraction) of the common length is not recycled but is an error, so that a mistyped argument length does not quietly produce a plausible-looking result. The resulting XPath expression can be given a prefix which determines the scope of the expression. The default prefix determines the scope to be the node itself and all descendants of the node. Most commonly the prefix is either the default or "", unless it is known what scope a particular XPath expression should have.

A selector starting with the :scope pseudo-class is anchored at the node the expression is evaluated from: the prefix argument is ignored and the expression begins with the XPath self axis instead. For example, ":scope > a" translates to "self:*/a", matching only the a children of the queried node, and a bare ":scope" translates to "self:*", matching the queried node itself. :scope anywhere else in a selector (after a combinator, or within a functional pseudo-class such as :is() or :has()) cannot be expressed in XPath 1.0 and is an error.

A type selector carrying no namespace prefix, such as "p", becomes an XPath name test and so matches elements in no namespace only. That is true wherever the name appears in the selector, so ":is(p)", ":not(p)" and ":has(p)" match exactly the elements "p" itself does. An element that is in a namespace, a default namespace declared with xmlns included, has to be selected through a prefix, as in "d|p": prefixes are resolved through the namespace map supplied when the expression is evaluated (the ns argument of `xml_find_all` or `getNodeSet`), not through the prefix spelled in the document. Two forms need no such map: "*|p" matches p in any namespace, and "|p" is the explicit spelling of p in no namespace.

The of-type pseudo-classes (:first-of-type, :last-of-type, :only-of-type, :nth-of-type() and :nth-last-of-type()) are only supported when their compound selector names an element, as in "p:first-of-type". Applied to the universal selector, as in "*:first-of-type", they would have to compare each sibling's name against the matched element's own name, which XPath 1.0 cannot express, so the translation is an error. The Python 'cssselect' library, from which selectr is ported, has the same limitation.

The Selectors 4 column combinator ("a || b") and the column pseudo-classes :nth-col() and :nth-last-col() are also not supported: which column a cell belongs to depends on table-layout arithmetic (colspan/rowspan carry-over) that XPath 1.0 cannot express. Both are rejected with an error.

:empty deliberately keeps the Selectors 3 semantics that all current browsers implement: an element containing only white space, such as <p> </p>, does not match. (The Selectors 4 specification loosened :empty to also match white-space-only elements, but no browser has shipped that change.)

:lang() ranges are matched as RFC 4647 language ranges, so a wildcard may appear as a whole range (:lang(*)), as a trailing subtag (:lang(en-*)), or in a non-trailing position (:lang(*-CH), :lang(de-*DE); quoted or not). The html and xhtml translators implement RFC 4647 extended filtering for any range naming more than one subtag, approximated from the nearest language-attributed ancestor: a wildcard, explicit or not, may skip any subtag, so :lang(de-DE) - with no wildcard at all - matches lang="de-Latn-DE". The first subtags of range and tag are always paired, though, so a leading wildcard *consumes* the tag's primary subtag rather than skipping over it: :lang(*-CH) matches lang="de-CH" and lang="fr-Latn-CH", but neither lang="ch" nor lang="ch-DE". A single-subtag range (:lang(en), :lang(en-*)) is a plain prefix test, as before. The generic translator has only XPath's lang() function, which does Selectors 3 |=-style prefix matching and cannot express extended filtering: a non-trailing wildcard is rejected there with an error rather than silently mis-matching, and a multi-subtag range with no wildcard (e.g. :lang(de-DE)) is matched as a prefix only - it does *not* skip subtags the way the html/xhtml translators do.

Every range must be an RFC 4647 *extended language range*: each subtag is either a whole * or one to eight alphanumeric characters (letters only, for the first), so an empty, over-long or non-alphanumeric subtag, or a * glued to a subtag, is rejected with an error naming the range (:lang(en-), :lang(--x), :lang(en*), :lang(de-*--de)). The specification says such a range simply matches nothing while leaving the selector valid; XPath has no never-matching form that survives being combined with the rest of the expression, so the range is refused rather than translated into a well-formed one that would select the wrong elements. An empty item of the comma-separated list (:lang(en,)) is likewise an error.

:lang("") matches an element whose content language is not tagged at all: no lang/xml:lang (as applicable to the translator) anywhere in its ancestor-or-self chain, or only an empty one. This holds for every translator.

Which attribute supplies that language differs by translator: the html translator reads lang, while the xhtml translator reads xml:lang or lang, preferring xml:lang where an element carries both, as the HTML language determination does. The generic translator uses XPath's lang() function, which is defined in terms of xml:lang alone. With every translator the language comes from the nearest ancestor-or-self that declares one, and an empty value there resets the language to unknown.

:dir() translates to a never-matching expression with every translator, including html: an element's resolved directionality also depends on dir="auto", bdi, and form-control rules that a static document cannot answer, so unlike :lang() it is not approximated from ancestor attributes.

The translator used is usually unnecessary to specify as the default is sufficient for most cases. However, it is of use when creating expressions relating to (X)HTML pseudo elements and languages. In particular it qualifies a number of pseudo-classes - :checked, :default, :disabled, :enabled, :link, :optional, :placeholder-shown, :read-only, :read-write and :required - to apply only to relevant (X)HTML elements, identified by local name regardless of namespace so that, for example, "*|input:disabled" and "d1|input:disabled" apply :disabled exactly as "input:disabled" does on an unnamespaced HTML document. See [selectors](#) for exactly which elements and attributes each of these matches.

When the translator is set to html, all element and attribute names will be converted to lower case (A-Z only, as an HTML parser folds them), and the attributes HTML defines as ASCII case-insensitive - type, rel, lang and the rest of the list in [selectors](#) - also compare their *values* without regard to case. Both are removed when the translator is xhtml (or the default generic translator), neither of which serves HTML documents.

Value

A character vector of XPath expressions.

Errors

Every error `css_to_xpath` raises is a classed condition inheriting `selectr_error` (itself an error), so a caller can catch the whole family with one handler, or narrow to a specific class when it needs to react differently:

`selectr_parse_error` selector is malformed CSS, e.g. "div>" with nothing following the combinator. Carries selector (the offending input) and pos, the 1-based character position within selector of the token the message names (NULL if no position applies).

`selectr_translation_error` selector is valid CSS but cannot be expressed in XPath 1.0 by the requested translator, e.g. :scope nested inside :is(), or an unknown pseudo-class such as ":frobnicate". Carries selector and feature, a short description of the unsupported construct, and - for a construct that can be located, which is every unknown pseudo-class, misplaced :scope, pseudo-element and :lang() argument - pos and column, the 1-based character position of the construct within selector and within its line. Both are NULL for the two failures no single position describes: a namespace prefix that is not an XPath name, and an of-type pseudo-class on the universal selector. feature is the CSS that spells the construct, except for that namespace-prefix failure, which no CSS spells and which reads as the phrase the message uses, e.g. "a namespace prefix that is not an XPath name (`1ns`)".

`selectr_argument_error` An R-level argument is invalid: the wrong type, length, or an NA or unrecognised value where one is not allowed. Carries no extra fields.

```

tryCatch(
  css_to_xpath("div >"),
  selectr_parse_error = function(e) {
    cat(conditionMessage(e), "\n")
    cat("Position:", e$pos, "\n")
  }
)

```

Author(s)

Simon Potter

References

CSS Selectors Level 4 <https://www.w3.org/TR/selectors-4/>, XPath <https://www.w3.org/TR/xpath/>.

See Also

[selectors](#) for the full selector-support reference; [querySelectorAll](#), which propagates the same conditions.

Examples

```

css_to_xpath(".testclass")
css_to_xpath("#testid", prefix = "")
css_to_xpath("#testid .testclass")
css_to_xpath(":scope > .testclass")
css_to_xpath(":checked", translator = "html")

# The selectr_parse_error, selectr_translation_error and
# selectr_argument_error conditions (see 'Errors' below) all inherit
# 'selectr_error', so callers can catch the family or a specific class.
tryCatch(
  css_to_xpath("div >"),
  selectr_parse_error = function(e) {
    cat(conditionMessage(e), "\n")
    cat("Position:", e$pos, "\n")
  }
)

```

Description

The purpose of these functions is to mimic the functionality of the `querySelector` and `querySelectorAll` functions present in Internet browsers. This is so we can succinctly query an XML tree for nodes matching a CSS selector.

Namespaced functions `querySelectorNS` and `querySelectorAllNS` are also provided to search relative to a given namespace.

Usage

```
querySelector(doc, selector, ns = NULL, ...)
querySelectorAll(doc, selector, ns = NULL, ...)
querySelectorNS(doc, selector, ns,
                prefix = "descendant-or-self::", ...)
querySelectorAllNS(doc, selector, ns,
                  prefix = "descendant-or-self::", ...)
```

Arguments

<code>doc</code>	The XML document, node, or set of nodes to be evaluated against.
<code>selector</code>	A selector used to query <code>doc</code> . This must be a single character string.
<code>ns</code>	The namespaces that the query will be filtered to. This is a named list or vector whose name is the prefix a selector uses (the <code>svg</code> in <code>"svg g"</code>), and whose value is the namespace URI that prefix stands for. Each name must be a valid XML NCName: a letter or <code>_</code> followed by letters, digits, <code>.</code> , <code>-</code> or <code>_</code> , in any script, and no colon. This can be ignored for the un-namespaced functions, where a zero-length <code>ns</code> (<code>character(0)</code> or <code>list()</code>) additionally means no namespaces at all.
<code>prefix</code>	The prefix to apply to the resulting XPath expression. The default or <code>""</code> are most commonly used.
<code>...</code>	Parameters to be passed onto <code>css_to_xpath</code> .

Details

The `querySelectorNS` and `querySelectorAllNS` functions are convenience functions for working with namespaced documents. They filter out all content that does not belong within the given namespaces. Note that when searching for particular elements in a selector, they must have a namespace prefix, e.g. `"svg|g"`. The filter is relative to `doc`, so like the un-namespaced functions these search a node's own subtree rather than the whole document. A selector starting with `:scope` replaces the filter altogether (see below); such a selector is namespaced by its own prefixes, e.g. `":scope > svg|g"`.

The namespace argument, `ns`, is simply passed on to `getNodeSet` or `xml_find_all` if it is necessary to use a namespace present within the document. This can be ignored for content lacking a namespace, which is usually the case when using `querySelector` or `querySelectorAll`.

For `querySelector` and `querySelectorAll`, leaving `ns` as `NULL` on an **xml2** document means the document's own namespace map is used, which `xml_ns` builds by walking the whole document on every query. Passing a zero-length `ns`, `character(0)` or `list()`, skips that lookup and queries with no namespaces, which is worth doing on a large document known to be un-namespaced. It is an error for the namespaced functions, which have nothing to filter to without a namespace.

A selector's bare element names match elements in no namespace, `":is(p)"` and `":has(p)"` exactly as `"p"` itself does, so an element in a default namespace has to be reached through a prefix. With **xml2** the document's own prefixes are used when `ns` is not given, and `xml_ns` names a default namespace `d1`, making `"d1|p"` the selector for those elements.

Queries may be chained: as well as a document or a single node, `doc` may be a set of nodes, i.e. an **xml2** `xml_nodeset` or an **XML** `XMLNodeSet`, as returned by `querySelectorAll`. The selector is then evaluated from each node of the set in turn, so a relative selector such as `":scope > a"` applies per node. A node that matches from more than one node of the set is returned only once, at the position it first matched. An **xml2** `xml_missing` (the result of a failed `xml_find_first`) is also accepted, and yields no matches rather than an error.

Selectors are translated with the generic (XML) translator unless a `translator` argument is given to be passed on to `css_to_xpath`, with one exception: a document parsed as HTML by `htmlParse` or `read_html` is queried with the `html` translator, so that element and attribute names are matched case-insensitively and the pseudo-classes that depend on HTML semantics (`:checked`, `:disabled`, `:link`, `:lang()` via the `lang` attribute, ...) work as they do in a browser. Passing `translator` explicitly overrides this for either kind of document.

The document is recognised however the query starts, so a chain of queries beginning at an HTML document keeps the `html` translator when it continues from one of the document's nodes or from a set of them.

A selector starting with the `:scope` pseudo-class is anchored at the queried node itself: `querySelectorAll(node, ":scope > a")` returns only the `a` children of `node`, where `querySelectorAll(node, "a")` would return all of its `a` descendants. `:scope` after a combinator or within a functional pseudo-class is an error (it cannot be expressed in XPath 1.0).

When `doc` is a whole document rather than a node, the queried node is taken to be the document's root element, so a bare `:scope` matches that root element and `":scope > x"` matches its `x` children. This differs from a browser's `document.querySelector()`, where `:scope` on a document refers to the document itself: a bare `:scope` matches nothing there (the document is not an element), while `":scope > html"` matches the root element. To query starting from the root element itself rather than the document, pass the root node (e.g. `xmlRoot` or `xml_root`) as `doc` instead of the document.

Value

For `querySelector`, the result is a single node that represents the first matched node from a selector. If no matching nodes are found, `NULL` is returned.

For `querySelectorAll`, the result is a list of XML nodes. This list may be empty in the case that no match is found. The list is of the same type as the input document's package uses, so querying an `xml_nodeset` gives an `xml_nodeset` and querying an `XMLNodeSet` gives an `XMLNodeSet`.

The `querySelectorNS` and `querySelectorAllNS` functions return the same type of content as their un-namespaced counterparts.

Errors

These functions propagate the same `selectr_parse_error` and `selectr_translation_error` conditions that `css_to_xpath` raises for a malformed or unsupported selector (see `?css_to_xpath` for their fields), plus `selectr_argument_error` for a bad R-level argument: `doc` that is not an **XML** or **xml2** document, node, or node set; `selector` that is not a single character string; or `ns`

that is not a named list or named character vector of non-empty strings whose names are valid XML names (or, for `querySelectorNS` and `querySelectorAllNS`, a missing or zero-length `ns`).

Author(s)

Simon Potter

References

CSS Selectors Level 4 <https://www.w3.org/TR/selectors-4/>, XPath <https://www.w3.org/TR/xpath/>, `querySelectorAll` <https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll> and <https://dom.spec.whatwg.org/#dom-parentnode-queryselectorall>.

See Also

[css_to_xpath](#), whose ‘Errors’ section documents the condition classes’ fields; [selectors](#) for the full selector-support reference.

Examples

```
# All three selectr_error classes (see 'Errors' below) propagate
# from these functions; selectr_argument_error also covers a 'doc'
# that is not an XML or xml2 document, node, or node set.
tryCatch(
  querySelectorAll("not a document", "a"),
  selectr_argument_error = function(e) cat(conditionMessage(e), "\n")
)

# The XML and xml2 packages are both optional (Suggests), so each demo
# below is guarded with requireNamespace() and runs only when that
# package is installed.

# Demo for working with the XML package
if (requireNamespace("XML", quietly = TRUE)) {
  exdoc <- XML::xmlParse('<a><b class="aclass"/><c id="anid"/></a>')
  querySelector(exdoc, "#anid") # Returns the matching node
  querySelector(exdoc, ".aclass") # Returns the matching node
  querySelector(exdoc, "b, c") # First match from grouped selection
  querySelectorAll(exdoc, "b, c") # Grouped selection
  querySelectorAll(exdoc, "b") # A list of length one
  querySelector(exdoc, "d") # No match
  querySelectorAll(exdoc, "d") # No match

  # Queries can be chained, the second search running from each node
  # matched by the first
  querySelectorAll(querySelectorAll(exdoc, "a"), "c")

  # Read in a document where two namespaces are being set:
  # SVG and MathML
  svgdoc <- XML::xmlParse(system.file("demos/svg-mathml.svg",
    package = "selectr"))
  # Search for <script/> elements in the SVG namespace
```

```

querySelectorNS(svgdoc, "svg|script",
                c(svg = "http://www.w3.org/2000/svg"))
querySelectorAllNS(svgdoc, "svg|script",
                  c(svg = "http://www.w3.org/2000/svg"))
# MathML content is *within* SVG content,
# search for <mtext> elements within the MathML namespace
querySelectorNS(svgdoc, "math|mtext",
                c(math = "http://www.w3.org/1998/Math/MathML"))
querySelectorAllNS(svgdoc, "math|mtext",
                  c(math = "http://www.w3.org/1998/Math/MathML"))
# Search for *both* SVG and MathML content
querySelectorAllNS(svgdoc, "svg|script, math|mo",
                  c(svg = "http://www.w3.org/2000/svg",
                    math = "http://www.w3.org/1998/Math/MathML"))
}

# Demo for working with the xml2 package
if (requireNamespace("xml2", quietly = TRUE)) {
  exdoc <- xml2::read_xml('<a><b class="aclass"><c id="anid"></a>')
  querySelector(exdoc, "#anid") # Returns the matching node
  querySelector(exdoc, ".aclass") # Returns the matching node
  querySelector(exdoc, "b, c") # First match from grouped selection
  querySelectorAll(exdoc, "b, c") # Grouped selection
  querySelectorAll(exdoc, "b") # A nodeset of length one
  querySelector(exdoc, "d") # No match
  querySelectorAll(exdoc, "d") # No match

  # Queries can be chained, the second search running from each node
  # matched by the first
  querySelectorAll(querySelectorAll(exdoc, "a"), "c")

  # Read in a document where two namespaces are being set:
  # SVG and MathML
  svgdoc <- xml2::read_xml(system.file("demos/svg-mathml.svg",
                                       package = "selectr"))

  # Search for <script> elements in the SVG namespace
  querySelectorNS(svgdoc, "svg|script",
                  c(svg = "http://www.w3.org/2000/svg"))
  querySelectorAllNS(svgdoc, "svg|script",
                    c(svg = "http://www.w3.org/2000/svg"))
  # MathML content is *within* SVG content,
  # search for <mtext> elements within the MathML namespace
  querySelectorNS(svgdoc, "math|mtext",
                  c(math = "http://www.w3.org/1998/Math/MathML"))
  querySelectorAllNS(svgdoc, "math|mtext",
                    c(math = "http://www.w3.org/1998/Math/MathML"))
  # Search for *both* SVG and MathML content
  querySelectorAllNS(svgdoc, "svg|script, math|mo",
                    c(svg = "http://www.w3.org/2000/svg",
                      math = "http://www.w3.org/1998/Math/MathML"))
}

```

Description

A reference table of every combinator, simple selector, attribute operator and pseudo-class selectr recognises: whether it is supported, restricted to the html/xhtml translators, never matches (a static-document limitation), or is rejected as an error, plus the XPath a representative selector translates to. This complements the prose in [css_to_xpath](#), which explains *why* the divergences from CSS Selectors Level 4 below exist; this page is the flat list to check "is X supported?" against.

Every example on this page is exercised by tests/testthat/test-selectors-reference.R against a live [css_to_xpath](#) call, and that test also asserts every xpath_*_pseudo, xpath_*_function and xpath_*_combinator method of GenericTranslator and HTMLTranslator is represented on this page, so this table cannot silently drift from the translators' code.

Combinators

Selector	Meaning	Example XPath ("e ? f", generic)
e f	descendant	descendant-or-self::e//f
e > f	child	descendant-or-self::e/f
e + f	direct adjacent sibling	descendant-or-self::e/following-sibling::*[1][self::f]
e ~ f	indirect (general) sibling	descendant-or-self::e/following-sibling::f
e f	column (Selectors 4)	error - not supported, see below

Simple selectors

Selector	Meaning	Example XPath
*	universal	descendant-or-self::*
e	type (no namespace)	descendant-or-self::e
.class	class	...[contains(concat(' ', normalize-space(@class), ' '), ' class ')]
#id	ID	descendant-or-self::*[@id = 'id']

Class matching splits @class on XML whitespace (space/tab/CR/LF) via `normalize-space()`; HTML's own "set of space-separated tokens" also treats U+000C form feed as a separator, which this does not - negligible in practice, since form feed in a class attribute is vanishingly rare.

Attribute selectors

Selector	Meaning	Example XPath ("[attr ? val]")
[attr]	has attribute	descendant-or-self::*[@attr]

[attr=val]	equals	descendant-or-self::*[@attr = 'val']
[attr~=val]	includes a whitespace-separated token	...[contains(concat(' ', normalize-space(@attr), ' '), 'val')]
[attr =val]	equals, or a "val-" prefix	...[@attr = 'val' or starts-with(@attr, 'val-')]
[attr^=val]	starts with	descendant-or-self::*[starts-with(@attr, 'val')]
[attr\$=val]	ends with	...[substring(@attr, string-length(@attr)-2) = 'val']
[attr*=val]	contains substring	descendant-or-self::*[contains(@attr, 'val')]

Every operator above accepts a trailing Selectors 4 case-sensitivity flag, *i* (ASCII case-insensitive) or *s* (case-sensitive, the default and so a no-op): [attr=val *i*] translates to ...[translate(@attr, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'abcdefghijklmnopqrstuvwxyz') = 'val']. Any other flag is a parse error.

Under the `html` translator, the attributes HTML lists as ASCII case-insensitive (type, rel, lang, hreflang, dir, media, method, target, checked, disabled, readonly, selected, multiple, shape, scope, align, charset, http-equiv, enctype, accept and the rest of HTML's "Case-sensitivity of selectors" list) match their values that way with no flag written, as they do in a browser: `input[type=radio]` selects `<input type="RADIO">`. An explicit *s* restores the exact comparison. `class`, `id`, `href`, `name`, `data-*` and every namespaced attribute keep it, and so does every attribute under the generic and `xhtml` translators, which serve XML rather than HTML documents.

Structural pseudo-classes

Selector	Status	Example XPath
:root	supported	descendant-or-self::*[not(parent)]
:first-child	supported	...[not(preceding-sibling::*[1])]
:last-child	supported	...[not(following-sibling::*[1])]
:only-child	supported	preceding- and following-sibling counts
e:first-of-type	supported (needs a named element)	...e[not(preceding-sibling::e[1])]
e:last-of-type	supported (needs a named element)	...e[not(following-sibling::e[1])]
e:only-of-type	supported (needs a named element)	preceding- and following-sibling e count
*:first-of-type etc.	error	needs each sibling's own name; not exp
:nth-child(An+B)	supported	:nth-child(2n+1) gives ...[count(p
:nth-child(An+B of S)	supported	preceding-siblings filtered to S, plus sel
:nth-last-child()	supported	as :nth-child(), counting from the en
e:nth-of-type(), e:nth-last-of-type()	supported (needs a named element)	as above, restricted to siblings named e
:empty	supported, Selectors 3 semantics	...[not(*) and not(string-length(
:scope	supported, leading position only	self::* (replaces the prefix); errors e

Selector-list pseudo-classes

Selector	Meaning	Example XPath
:not(e)	none of the arguments match	descendant-or-self::*[not(self::e)]
:is(e, f) (alias :matches())	any argument matches	descendant-or-self::*[self::e or self::f]

:where(e, f)	any argument matches (zero specificity)	same XPath as :is()
:has(> e)	a descendant/relative match exists	descendant-or-self::*[child::e]

Each accepts a full selector list, but a :scope inside any of them is an error (see :scope above), and the column combinator inside them is likewise unsupported.

Linguistic and directionality pseudo-classes

Selector	Status	Notes
:lang(range)	supported, translator-dependent	generic: XPath lang(), prefix match only. html/xhtml: RFC 4647 extension
:lang("")	supported, every translator	matches an element with no content language anywhere in its ancestor-or-self::*
:dir()	never matches, every translator	descendant-or-self::*[0]; directionality needs a live DOM (dir="auto")

Link and interaction-state pseudo-classes

Selector	generic	html / xhtml
:link, :any-link	never matches	matches a and area elements with an href
:visited	never matches	never matches (no browser history in a static document)
:hover, :active, :focus, :focus-within, :focus-visible	never matches	never matches (runtime UI state)
:target, :target-within	never matches	never matches (needs the document's URL)
:local-link	never matches	never matches (needs the document's URL)

"Never matches" translates to descendant-or-self::*[0]: valid CSS, always zero results, rather than an error. In a larger compound the always-false 0 absorbs the compound's other conditions, which cannot change the outcome, so "a.external:visited" translates to descendant-or-self::a[0] as well.

HTML form-state pseudo-classes

These are only meaningfully supported by the html and xhtml translators (under generic they never match, listed above as the general runtime-state case). Every one matches by local name regardless of namespace, so "*|input:disabled" works the same as "input:disabled" on an unnamespaced document. :enabled and :disabled match only the elements listed below - in particular a hyperlink is not :enabled; use :link or :any-link for links.

Selector	Elements and condition (html/xhtml)
:enabled / :disabled	button, input, select, textarea, optgroup, option, fieldset; a disabled ancestor fieldset disallows
:checked	checked checkbox/radio inputs and selected options; does <i>not</i> infer the implicit default selection
:required / :optional	input of a type that takes required (every type but hidden, range, color, submit, image, reset, button)
:read-write	an input of a type that takes readonly (every type but hidden, color, checkbox, radio, file, submit, button)
:read-only	the negation of :read-write (matches everything else, e.g. a checkbox or a plain div)
:placeholder-shown	textarea, or an input of a type that takes placeholder (every type but hidden, checkbox, radio, submit, button)

:default a selected option, a checked checkbox/radio, or the first submit button in its nearest ancestor for

Because HTML’s type is an enumerated attribute, these match its keywords ASCII case-insensitively (<input type="RADIO"> is :checked). An input with no type, or with an unrecognised one, is in the text state, as it is for an HTML parser.

Column combinator and pseudo-classes (unsupported)

The Selectors 4 column combinator (a || b) and the column pseudo-classes :nth-col() / :nth-last-col() are rejected with an error: which column a cell belongs to depends on colspan/rowspan table-layout arithmetic that XPath 1.0 cannot express.

Namespaces

Selector	Meaning	Example XPath ("? p")
p	p in no namespace	descendant-or-self::p
d p	p in the namespace prefix d resolves to via the ns map	descendant-or-self::d:p
* p	p in any namespace	descendant-or-self::*[local-name() = 'p']
p	p in no namespace, spelled explicitly	descendant-or-self::p

Prefixes such as d above are resolved through the ns argument passed to `xml_find_all / getNodeSet` at query time, not through whatever prefix the document itself uses; see `querySelectorAll`. A prefix is written into the generated XPath as it stands, so it has to be a name XPath can parse (an XML NCName, which is not restricted to ASCII); one that is not, such as the escaped `\31 ns|div`, is rejected with an error rather than compared against the document’s own prefix. An escaped `*` (`\2a|div`) is such a prefix too: only the delimiter `*` of `*|p` above is the any-namespace wildcard, and a prefix spelled by an identifier is one no `@namespace` rule could have bound. Local names carry no such restriction: one that cannot be written as a name test is compared with `local-name()` instead, e.g. `d|\31` becomes `d:*[local-name() = '1']`. `HTMLTranslator` additionally lower-cases every element and attribute name - folding A-Z only, as an HTML parser does, so a non-ASCII name is left as written - including namespaced ones, so `svg|linearGradient` becomes `svg:lineargradient`, which matches `libxml2`’s HTML parser but would be wrong against a tree that restores camelCase SVG/MathML names (browsers, `html5ever`).

Divergences from CSS Selectors Level 4

- 1. `:empty` keeps Selectors 3 semantics: an element containing only white space, e.g. `<p> </p>`, does not match. Selectors 4 loosened this to also match white-space-only content, but no browser has shipped that change, so it is treated as not implemented, tracking browser behaviour rather than the spec text.
- 2. `:checked` tests only `@checked/@selected`. It does not infer the implicit selectedness of an option with no selected attribute anywhere in its select (the first option is selected by default), nor a radio group’s mutual exclusivity - both need a live DOM to resolve.
- 3. `HTMLTranslator` lower-cases foreign-content element and attribute names unconditionally, targeting `libxml2`-style HTML trees (see "Namespaces" above); an HTML5 parser that restores camelCase SVG/MathML names would disagree.

4. Class/token matching (`.foo`, `[attr~=val]`) does not treat U+000C form feed as whitespace, unlike HTML's ASCII whitespace definition (see "Simple selectors" above).

Author(s)

Simon Potter

References

CSS Selectors Level 4 <https://www.w3.org/TR/selectors-4/>, XPath <https://www.w3.org/TR/xpath/>.

See Also

[css_to_xpath](#) for the full prose explanation of each divergence above and the error classes raised for unsupported selectors; [querySelectorAll](#) for namespace and chaining semantics when querying a document.

Index

`css_to_xpath`, [2](#), [7](#), [8](#), [10](#), [12](#), [14](#)

`getNodeSet`, [3](#), [6](#), [13](#)

`htmlParse`, [7](#)

`querySelector` (`querySelectorAll`), [5](#)

`querySelectorAll`, [5](#), [5](#), [13](#), [14](#)

`querySelectorAllNS` (`querySelectorAll`), [5](#)

`querySelectorNS` (`querySelectorAll`), [5](#)

`read_html`, [7](#)

`selectors`, [2](#), [4](#), [5](#), [8](#), [10](#)

`xml_find_all`, [3](#), [6](#), [13](#)

`xml_find_first`, [7](#)

`xml_ns`, [6](#), [7](#)

`xml_root`, [7](#)

`xmlRoot`, [7](#)