W3C Candidate Recommendation

W3C

CSS Custom Properties for Cascading Variables Module Level 1

W3C Candidate Recommendation, 03 December 2015

This version:
http://www.w3.org/TR/2015/CR-css-variables-1-20151203/
Latest version:
http://www.w3.org/TR/css-variables-1/
Editor's Draft:
http://dev.w3.org/csswg/css-variables/
Previous Versions:
http://www.w3.org/TR/2014/WD-css-variables-1-20140506/
http://www.w3.org/TR/2013/WD-css-variables-1-20130620/
http://www.w3.org/TR/2013/WD-css-variables-20130312/
http://www.w3.org/TR/2012/WD-css-variables-20120410/
Feedback:
www-style@w3.org with subject line “[css-variables] … *message topic* …” (archives)
Test Suite:
http://test.csswg.org/suites/css-variables-1_dev/nightly-unstable/

Editor:
Tab Atkins Jr. (Google)

Abstract

This module introduces cascading variables as a new primitive value type that is accepted by all CSS properties, and custom properties for defining them.

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

Status of this document

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

This document was produced by the CSS Working Group (part of the Style Activity) as a Candidate Recommendation. This document is intended to become a W3C Recommendation. This document will remain a Candidate Recommendation at least until 1 June 2016 in order to ensure the opportunity for wide review.

A preliminary implementation report is available.

The (archived) public mailing list www-style@w3.org (see instructions) is preferred for discussion of this specification. When sending e-mail, please put the text “css-variables” in the subject, preferably like this: “[css-variables] …*summary of comment*…”

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

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

This document is governed by the 1 September 2015 W3C Process Document.

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

Table of Contents

§ 1. Introduction

This section is not normative.

Large documents or applications (and even small ones) can contain quite a bit of CSS. Many of the values in the CSS file will be duplicate data; for example, a site may establish a color scheme and reuse three or four colors throughout the site. Altering this data can be difficult and error-prone, since it's scattered throughout the CSS file (and possibly across multiple files), and may not be amenable to Find-and-Replace.

This module introduces a family of custom author-defined properties known collectively as custom properties, which allow an author to assign arbitrary values to a property with an author-chosen name, and the ‘var()’ function, which allow an author to then use those values in other properties elsewhere in the document. This makes it easier to read large files, as seemingly-arbitrary values now have informative names, and makes editing such files much easier and less error-prone, as one only has to change the value once, in the custom property, and the change will propagate to all uses of that variable automatically.

§ 2. Defining Custom Properties: the ‘--*’ family of properties

This specification defines an open-ended set of properties called custom properties, which, among other things, are used to define the substitution value of ‘var()’ functions.

Name:	‘--*’
Value:	<declaration-value>
Initial:	(nothing, see prose)
Applies to:	all elements
Inherited:	yes
Percentages:	n/a
Media:	all
Computed value:	specified value with variables substituted (but see prose for "invalid variables")
Animatable:	no

A ***custom property*** is any property whose name starts with two dashes (U+002D HYPHEN-MINUS), like ‘--foo’. The ***‘<custom-property-name>’*** production corresponds to this: it's defined as any valid identifier that starts with two dashes. Custom properties are solely for use by authors and users; CSS will never give them a meaning beyond what is presented here.

EXAMPLE 1
Custom properties define variables, referenced with the ‘var()’ notation, which can be used for many purposes. For example, a page that consistently uses a small set of colors in its design can store the colors in custom properties and use them with variables:

```
:root {
  --main-color: #06c;
  --accent-color: #006;
}
/* The rest of the CSS file */
#foo h1 {
  color: var(--main-color);
}
```

The naming provides a mnemonic for the colors, prevents difficult-to-spot typos in the color codes, and if the theme colors are ever changed, focuses the change on one simple spot (the custom property value) rather than requiring many edits across all stylesheets in the webpage.

Unlike other CSS properties, custom property names are case-sensitive.

EXAMPLE 2
While both ‘--foo’ and --FOO are valid, they are distinct properties - using ‘var(--foo)’ will refer to the first one, while using ‘var(--FOO)’ will refer to the second.

Custom properties are **not** reset by the ‘all’ property. We may define a property in the future that resets all variables.

The CSS-wide keywords can be used in custom properties, with the same meaning as in any another property.

Note: That is, they're interpreted at cascaded-value time as normal, and are not preserved as the custom property's value, and thus are not substituted in by the corresponding variable.

Note: While this module focuses on the use of custom properties with the ‘var()’ function to create “variables”, they can also be used as actual custom properties, parsed by and acted on by script. It's expected that the CSS Extensions spec [CSS-EXTENSIONS] will expand on these use-cases and make them easier to do.

§ 2.1. Custom Property Value Syntax

The allowed syntax for custom properties is extremely permissive. The <declaration-value> production matches *any* sequence of one or more tokens, so long as the sequence does not contain <bad-string-token>, <bad-url-token>, unmatched <)-token>, <]-token>, or <}-token>, or top-level <semicolon-token> tokens or <delim-token> tokens with a value of "!".

In addition, if the value of a custom property contains a ‘var()’ reference, the ‘var()’ reference must be valid according to the specified ‘var()’ grammar. If not, the custom property is invalid and must be ignored.

Note: This definition, along with the general CSS syntax rules, implies that a custom property value never includes an unmatched quote or bracket, and so cannot have any effect on larger syntax constructs, like the enclosing style rule, when reserialized.

Note: Custom properties can contain a trailing ‘!important’, but this is automatically removed from the property's value by the CSS parser, and makes the custom property "important" in the CSS cascade. In other words, the prohibition on top-level "!" characters does not prevent ‘!important’ from being used, as the ‘!important’ is removed before syntax checking happens.

Note: While <declaration-value> must represent at least one token, that one token may be whitespace. This implies that ‘--foo: ;’ is valid, and the corresponding ‘var(--foo)’ call would have a single space as its substitution value, but ‘--foo:;’ is invalid.

EXAMPLE 3
For example, the following is a valid custom property:

```
--foo: if(x > 5) this.width = 10;
```

While this value is obviously useless as a *variable*, as it would be invalid in any normal property, it might be read and acted on by JavaScript.

The values of custom properties, and the values of ‘var()’ functions substituted into custom properties, are *case-sensitive*, and must be preserved in their original author-given casing. (Many CSS values are ASCII case-insensitive, which user agents can take advantage of by "canonicalizing" them into a single casing, but that isn't allowed for custom properties.)

The initial value of a custom property is an empty value; that is, nothing at all. This initial value has a special interaction with the ‘var()’ notation, which is explained in the section defining ‘var()’.

Custom properties are ordinary properties, so they can be declared on any element, are resolved with the normal inheritance and cascade rules, can be made conditional with ‘@media’ and other conditional rules, can be used in HTML's `style` attribute, can be read or set using the CSSOM, etc.

Notably, they can even be transitioned or animated, but since the UA has no way to interpret their contents, they always use the "flips at 50%" behavior that is used for any other pair of values that can't be intelligently interpolated. However, any custom property used in a ‘@keyframes’ rule becomes ***animation-tainted***, which affects how it is treated when referred to via the ‘var()’ function in an animation property.

EXAMPLE 4
This style rule:

```
:root {
  --header-color: #06c;
}
```

declares a custom property named ‘--header-color’ on the root element, and assigns to it the value "#06c". This property is then inherited to the elements in the rest of the document. Its value can be referenced with the ‘var()’ function:

```
h1 { background-color: var(--header-color); }
```

The preceding rule is equivalent to writing ‘background-color: #06c;’, except that the variable name makes the origin of the color clearer, and if ‘var(--header-color)’ is used on other elements in the document, all of the uses can be updated at once by changing the ‘--header-color’ property on the root element.

EXAMPLE 5
If a custom property is declared multiple times, the standard cascade rules help resolve it. Variables always draw from the computed value of the associated custom property on the same element:

```
:root { --color: blue; }
div { --color: green; }
#alert { --color: red; }
* { color: var(--color); }

<p>I inherited blue from the root element!</p>
<div>I got green set directly on me!</div>
<div id='alert'>
  While I got red set directly on me!
  <p>I'm red too, because of inheritance!</p>
</div>
```

EXAMPLE 6
A real-world example of custom property usage is easily separating out strings from where they're used, to aid in maintenance of internationalization:

```
:root,
:root:lang(en) {--external-link: "external link";}
:root:lang(de) {--external-link: "externer Link";}

a[href^="http"]::after {content: " (" var(--external-link) ")"}
```

The variable declarations can even be kept in a separate file, to make maintaining the translations simpler.

§ 2.2. Resolving Dependency Cycles

Custom properties are left almost entirely unevaluated, except that they allow and evaluate the ‘var()’ function in their value. This can create cyclic dependencies where a custom property uses a ‘var()’ referring to itself, or two or more custom properties each attempt to refer to each other.

For each element, create a directed dependency graph, containing nodes for each custom property. If the value of a custom property *prop* contains a ‘var()’ function referring to the property *var* (including in the fallback argument of ‘var()’), add an edge between *prop* and the *var*. Edges are possible from a custom property to itself. If there is a cycle in the dependency graph, all the custom properties in the cycle must compute to their initial value (which is a guaranteed-invalid value).

EXAMPLE 7
This example shows a custom property safely using a variable:

```
:root {
  --main-color: #c06;
  --accent-background: linear-gradient(to top, var(--main-color), white);
}
```

The ‘--accent-background’ property (along with any other properties that use ‘var(--main-color)’) will automatically update when the ‘--main-color’ property is changed.

EXAMPLE 8
On the other hand, this example shows an invalid instance of variables depending on each other:

```
:root {
  --one: calc(var(--two) + 20px);
  --two: calc(var(--one) - 20px);
}
```

Both ‘--one’ and ‘--two’ now compute to their initial value, rather than lengths.

It is important to note that custom properties resolve any ‘var()’ functions in their values at computed-value time, which occurs *before* the value is inherited. In general, cyclic dependencies occur only when multiple custom properties on the same element refer to each other; custom properties defined on elements higher in the element tree can never cause a cyclic reference with properties defined on elements lower in the element tree.

EXAMPLE 9
For example, given the following structure, these custom properties are **not** cyclic, and all define valid variables:

```
<one><two><three /></two></one>
one   { --foo: 10px; }
two   { --bar: calc(var(--foo) + 10px); }
three { --foo: calc(var(--bar) + 10px); }
```

The <one> element defines a value for ‘--foo’. The <two> element inherits this value, and additionally assigns a value to ‘--bar’ using the ‘foo’ variable. Finally, the <three> element inherits the ‘--bar’ value *after* variable substitution (in other words, it sees the value ‘calc(10px + 10px)’), and then redefines ‘--foo’ in terms of that value. Since the value it inherited for ‘--bar’ no longer contains a reference to the ‘--foo’ property defined on <one>, defining ‘--foo’ using the ‘var(--bar)’ variable is not cyclic, and actually defines a value that will eventually (when referenced as a variable in a normal property) resolve to ‘30px’.

§ 3. Using Cascading Variables: the ‘var()’ notation

The value of a custom property can be substituted into the value of another property with the ‘var()’ function. The syntax of ‘var()’ is:

```
var() = var( <custom-property-name> [, <declaration-value> ]? )
```

The ‘var()’ function can be used in place of any part of a value in any property on an element. The ‘var()’ function can not be used as property names, selectors, or anything else besides property values. (Doing so usually produces invalid syntax, or else a value whose meaning has no connection to the variable.)

The first argument to the function is the name of the custom property to be substituted. The second argument to the function, if provided, is a fallback value, which is used as the substitution value when the referenced custom property is invalid.

Note: The syntax of the fallback, like that of custom properties, allows commas. For example, ‘var(--foo, red, blue)’ defines a fallback of ‘red, blue’; that is, anything between the first comma and the end of the function is considered a fallback value.

If a property contains one or more ‘var()’ functions, and those functions are syntactically valid, the entire property's grammar must be assumed to be valid at parse time. It is only syntax-checked at computed-value time, after ‘var()’ functions have been substituted.

To ***substitute a var()*** in a property's value:

1. If the custom property named by the first argument to the ‘var()’ function is animation-tainted, and the ‘var()’ function is being used in the ‘animation’ property or one of its longhands, treat the custom property as having its initial value for the rest of this algorithm.
2. If the value of the custom property named by the first argument to the ‘var()’ function is anything but the initial value, replace the ‘var()’ function by the value of the corresponding custom property.
3. Otherwise, if the ‘var()’ function has a fallback value as its second argument, replace the ‘var()’ function by the fallback value. If there are any ‘var()’ references in the fallback, substitute them as well.
4. Otherwise, the property containing the ‘var()’ function is invalid at computed-value time.

Note: Other things can also make a property invalid at computed-value time.

EXAMPLE 10
The fallback value allows for some types of defensive coding. For example, an author may create a component intended to be included in a larger application, and use variables to style it so that it's easy for the author of the larger application to theme the component to match the rest of the app.

Without fallback, the app author must supply a value for every variable that your component uses. With fallback, the component author can supply defaults, so the app author only needs to supply values for the variables they wish to override.

```
/* In the component's style: */
.component .header {
  color: var(--header-color, blue);
}
.component .text {
  color: var(--text-color, black);
}

/* In the larger application's style: */
.component {
  --text-color: #080;
  /* header-color isn't set,
     and so remains blue,
     the fallback value */
}
```

EXAMPLE 11
For example, the following code incorrectly attempts to use a variable as a property name:

```
.foo {
  --side: margin-top;
  var(--side): 20px;
}
```

This is *not* equivalent to setting ‘margin-top: 20px;’. Instead, the second declaration is simply thrown away as a syntax error for having an invalid property name.

Similarly, you can't build up a single token where part of it is provided by a variable:

```
.foo {
  --gap: 20;
  margin-top: var(--gap)px;
}
```

Again, this is *not* equivalent to setting ‘margin-top: 20px;’ (a length). Instead, it's equivalent to ‘margin-top: 20 px;’ (a number followed by an ident), which is simply an invalid value for the ‘margin-top’ property. Note, though, that ‘calc()’ can be used to validly achieve the same thing, like so:

```
.foo {
  --gap: 20;
  margin-top: calc(var(--gap) * 1px);
}
```

‘var()’ functions are substituted at computed-value time. If a declaration, once all ‘var()’ functions are substituted in, is invalid, the declaration is invalid at computed-value time.

EXAMPLE 12
For example, the following usage is fine from a syntax standpoint, but results in nonsense when the variable is substituted in:

```
:root { --looks-valid: 20px; }
p { background-color: var(--looks-valid); }
```

Since ‘20px’ is an invalid value for ‘background-color’, this instance of the property computes to ‘transparent’ (the initial value for ‘background-color’) instead.

If the property was one that's inherited by default, such as ‘color’, it would compute to the inherited value rather than the initial value.

§ 3.1. Invalid Variables

When a custom property has its initial value, ‘var()’ functions cannot use it for substitution. Attempting to do so makes the declaration invalid at computed-value time, unless a valid fallback is specified.

A declaration can be ***invalid at computed-value time*** if it contains a ‘var()’ that references a custom property with its initial value, as explained above, or if it uses a valid custom property, but the property value, after substituting its ‘var()’ functions, is invalid. When this happens, the computed value of the property is either the property's inherited value or its initial value depending on whether the property is inherited or not, respectively, as if the property's value had been specified as the ‘unset’ keyword.

EXAMPLE 13
For example, in the following code:

```
:root { --not-a-color: 20px; }
p { background-color: red; }
p { background-color: var(--not-a-color); }
```

the <p> elements will have transparent backgrounds (the initial value for ‘background-color’), rather than red backgrounds. The same would happen if the custom property itself was unset, or contained an invalid ‘var()’ function.

Note the difference between this and what happens if the author had just written ‘background-color: 20px’ directly in their stylesheet - that would be a normal syntax error, which would cause the rule to be discarded, so the ‘background-color: red’ rule would be used instead.

Note: The invalid at computed-value time concept exists because variables can't "fail early" like other syntax errors can, so by the time the user agent realizes a property value is invalid, it's already thrown away the other cascaded values.

§ 3.2. Variables in Shorthand Properties

The use of ‘var()’ functions in shorthand properties presents some unique difficulties.

Ordinarily, the value of a shorthand property is separated out into its component longhand properties at parse time, and then the longhands themselves participate in the cascade, with the shorthand more-or-less discarded. If a ‘var()’ functions is used in a shorthand, however, one can't tell what values are meant to go where; it may in fact be impossible to separate it out at parse time, as a single ‘var()’ function may substitute in the value of several longhands at once.

To get around this, implementations must fill in longhands with a special, unobservable-to-authors ***pending-substitution value*** that indicates the shorthand contains a variable, and thus the longhand's value is pending variable substitution. This value must then be cascaded as normal, and at computed-value time, after ‘var()’ functions are finally substituted in, the shorthand must be parsed and the longhands must be given their appropriate values at that point.

Pending-substitution values must be serialized as the empty string, if an API allows them to be observed.

Similarly, while [CSSOM] defines that shorthand properties are serialized by appropriately concatenating the values of their corresponding longhands, shorthands that are specified with explicit ‘var()’ functions must serialize to the original, ‘var()’-containing value. For other shorthands, if any of the longhand subproperties for that shorthand have pending-substitution values then the serialized value of the shorthand must be the empty string.

§ 4. APIs

All custom property declarations have the *case-insensitive* flag set.

Note: Custom properties do not appear on a CSSStyleDeclaration object in camel-cased form, because their names may have both upper and lower case letters which indicate distinct custom properties. The sort of text transformation that automatic camel-casing performs is incompatible with this. They can still be accessed by their proper name via getPropertyValue()/etc.

§ 4.1. Serializing Custom Properties

Custom property names must be serialized with the casing as provided by the author.

Ordinarily, property names are restricted to the ASCII range and are ASCII case-insensitive, so implementations typically serialize the name lowercased.

§ 5. Changes since the May 6 2014 Last Call Working Draft

- Serialization of longhands when shorthand uses a variable was defined.
- Link to DOM's definition of "case-sensitive".
- Added example of using variables with ‘:lang()’ to do simple i18n.
- Clarified that usage of ‘var()’ in a custom property must be valid per the ‘var()’ grammar.

§ 6. Acknowledgments

Many thanks to several people in the CSS Working Group for keeping the dream of variables alive over the years, particularly Daniel Glazman and David Hyatt. Thanks to multiple people on the mailing list for helping contribute to the development of this incarnation of variables, particularly Brian Kardell, David Baron, François Remy, Roland Steiner, and Shane Stephens.

§ 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 14
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.

§ Requirements for Responsible Implementation of CSS

The following sections define several conformance requirements for implementing CSS responsibly, in a way that promotes interoperability in the present and future.

§ 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 property 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.

§ Implementations of CR-level Features

Once a specification reaches the Candidate Recommendation stage, implementers should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec, and should avoid exposing a prefixed variant of that feature.

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

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

§ CR exit criteria

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

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

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

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

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

§ Index

§ Terms defined by this specification

§ Terms defined by reference

§ References

§ Normative References

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

[CSS-ANIMATIONS-1]
CSS Animations Module Level 1 URL: http://www.w3.org/TR/css3-animations/

[CSS-CASCADE-4]
Elika Etemad; Tab Atkins Jr.. CSS Cascading and Inheritance Level 4. 8 September 2015. WD. URL: http://www.w3.org/TR/css-cascade-4/

[CSS-CONDITIONAL-3]
CSS Conditional Rules Module Level 3 URL: http://www.w3.org/TR/css3-conditional/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. CSS Syntax Module Level 3. 20 February 2014. CR. URL: http://www.w3.org/TR/css-syntax-3/

[CSS-VALUES]
Tab Atkins Jr.; Elika Etemad. CSS Values and Units Module Level 3. 11 June 2015. CR. URL: http://www.w3.org/TR/css-values/

[CSSOM]
Simon Pieters; Glenn Adams. CSS Object Model (CSSOM). 5 December 2013. WD. URL: http://www.w3.org/TR/cssom/

[DOM]
Document Object Model URL: http://www.w3.org/TR/dom/

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

[SELECTORS-4]
Selectors Level 4 URL: http://www.w3.org/TR/selectors4/

§ Informative References

[CSS-BACKGROUNDS-3]
CSS Backgrounds and Borders Module Level 3 URL: http://www.w3.org/TR/css3-background/

[CSS-COLOR-3]
CSS Color Module Level 3 URL: http://www.w3.org/TR/css3-color/

[CSS-EXTENSIONS]
Tab Atkins Jr.. CSS Extensions. ED. URL: http://dev.w3.org/csswg/css-extensions/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Media	Animatable	Computed value
‘--*’	<declaration-value>	(nothing, see prose)	all elements	yes	n/a	all	no	specified value with variables substituted (but see prose for "invalid variables")