CSS Custom Properties for Cascading Variables Module Level 1

W3C Candidate Recommendation Snapshot, 16 June 2022

▼ **More details about this document**

This version:
https://www.w3.org/TR/2022/CR-css-variables-1-20220616/

Latest published version:
https://www.w3.org/TR/css-variables-1/

Editor's Draft:
https://drafts.csswg.org/css-variables/

Previous Versions:
https://www.w3.org/TR/2021/CRD-css-variables-1-20211111/
https://www.w3.org/TR/2015/CR-css-variables-1-20151203/
https://www.w3.org/TR/2014/WD-css-variables-1-20140506/
https://www.w3.org/TR/2013/WD-css-variables-1-20130620/
https://www.w3.org/TR/2013/WD-css-variables-20130312/
https://www.w3.org/TR/2012/WD-css-variables-20120410/

History:
https://www.w3.org/standards/history/css-variables-1

Implementation Report:
https://wpt.fyi/results/css/css-variables

Test Suites:
http://test.csswg.org/suites/css-variables-1_dev/nightly-unstable/
https://wpt.fyi/results/css/css-variables/

Feedback:
CSSWG Issues Repository

Editor:
Tab Atkins Jr. (Google)

Suggest an Edit for this Spec:
GitHub Editor

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, 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 **Candidate Recommendation Snapshot** using the Recommendation track. Publication as a Candidate Recommendation does not imply endorsement by W3C and its Members. A Candidate Recommendation Snapshot has received wide review, is intended to gather implementation experience, and has commitments from Working Group members to royalty-free licensing for implementations. This document is intended to become a W3C Recommendation; it will remain a Candidate Recommendation at least until 16 August 2022 to gather additional feedback.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-variables" in the title, like this: "[css-variables] *...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.

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.

§ 1.1. Value Definitions

This specification follows the CSS property definition conventions from [CSS2] using the value definition syntax from [CSS-VALUES-3]. Value types not defined in this specification are defined in CSS Values & Units [CSS-VALUES-3]. Combination with other CSS modules may expand the definitions of these value types.

In addition to the property-specific values listed in their definitions, all properties defined in this specification also accept the CSS-wide keywords as their property value. For readability they have not been repeated explicitly.

§ 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:	the guaranteed-invalid value
Applies to:	all elements and all pseudo-elements (including those with restricted property lists)

Inherited:	yes
Percentages:	n/a
Computed value:	specified value with variables substituted, or the guaranteed-invalid value
Canonical order:	per grammar
Animation type:	discrete

User agents are expected to support this property on all media, including non-visual ones.

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 <dashed-ident> (a valid identifier that starts with two dashes), except '--' itself, which is reserved for future use by CSS. Custom properties are solely for use by authors and users; CSS will never give them a meaning beyond what is presented here.

▸ TESTS

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 *not* ASCII case-insensitive. Instead, custom property names are only equal to each other if they are identical to each other.

▸ TESTS

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.

Perhaps more surprisingly, '--foó' and '--foó' are distinct properties. The first is spelled with U+00F3 (LATIN SMALL LETTER O WITH ACUTE) while the second is spelled with an ASCII "o" followed by U+0301 (COMBINING ACUTE ACCENT), and the "identical to" relation uses direct codepoint-by-codepoint comparison to determine if two strings are equal, to avoid the complexities and pitfalls of unicode normalization and locale-specific collation.

Operating systems, keyboards, or input methods sometimes encode visually-identical text using different codepoint sequences. Authors are advised to choose variable names that avoid potential confusion or to use escapes and other means to ensure that similar appearing sequences are identical. See Section 2.3 in [CHARMOD-NORM] for examples.

EXAMPLE 3

Developers maintaining the following CSS might be confused why the test patch is red:

```
--fijord: red;
--fijord: green;
--fijord: blue;

.test {
  background-color: var(--fijord);
}
```

The reason is that the first custom property uses the character sequence LATIN SMALL LETTER F + LATIN SMALL LETTER I + LATIN SMALL LETTER J; the second, identical-looking one uses the character sequence LATIN SMALL LETTER F + LATIN SMALL LIGATURE IJ while the third uses the character sequence LATIN SMALL LIGATURE FI + LATIN SMALL LETTER J.

So the CSS contains three distinct custom properties, two of which are unused.

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.

▸ TESTS

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.

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.

▸ TESTS

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.

▸ TESTS

Animation-tainted is "infectious": custom properties which reference animation-tainted properties also become animation-tainted.

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(el) {--external-link: "εξωτερικός σύνδεσμος";}

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.

§ 2.1. 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 "!".

▸ TESTS

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.

▸ TESTS

EXAMPLE 7

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.)

▸ TESTS

Because custom properties can contain *anything*, there is no general way to know how to interpret what's inside of them (until they're substituted into a known property with 'var()'). Rather than have them *partially* resolve in some cases but not others, they're left completely unresolved; they're a bare stream of CSS tokens interspersed with 'var()' functions.

This has some knock-on implications. For example, relative URLs in CSS are resolved against the base URL of the stylesheet the value appears in. However, if a custom property like '--my-image: url(foo.jpg);' shows up in an `"/a/style.css"` stylesheet, it will not resolve into an absolute URL immediately; if that variable is later used in a *different* `"/b/style.css"` stylesheet like 'background: var(--my-image);', it will resolve *at that point* to `"/b/foo.jpg"`.

§ 2.2. Guaranteed-Invalid Values

The initial value of a custom property is a ***guaranteed-invalid value***. As defined in § 3 Using Cascading Variables: the var() notation, using 'var()' to substitute a custom property with this as its value makes the property referencing it invalid at computed-value time.

This value serializes as the empty string, but actually writing an empty value into a custom property, like '--foo: ;', is a valid (empty) value, not the guaranteed-invalid value. If, for whatever reason, one wants to manually reset a variable to the guaranteed-invalid value, using the keyword 'initial' will do this.

§ 2.3. 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 are invalid at computed-value time.

▸ TESTS

Note: Defined properties that participate in a dependency cycle either end up with invalid variables in their value (becoming invalid at computed-value time), or define their own cyclic handling (like 'font-size' using 'em' values). They do not compute to the guaranteed-invalid value like custom properties do.

EXAMPLE 8

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 9

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' are now invalid at computed-value time, and compute to the guaranteed-invalid 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.

▸ TESTS

EXAMPLE 10

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

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

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>? )
```

▸ TESTS

In an exception to the usual comma elision rules, which require commas to be omitted when they're not separating values, a bare comma, with nothing following it, must be treated as valid in 'var()', indicating an empty fallback value.

▸ TESTS

> Note: That is, 'var(--a,)' is a valid function, specifying that if the '--a' custom property is invalid or missing, the 'var()' should be replaced with nothing.

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.)

▸ TESTS

> 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.

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 value of the referenced custom property is the guaranteed-invalid value.

▸ TESTS

> 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.

EXAMPLE 12

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 */
}
```

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.

▸ TESTS

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 a property that is not animatable, 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.

▸ TESTS

Note that var() substitution takes place at the level of CSS tokens [css-syntax-3], not at a textual level; 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;
}
```

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);
}
```

▸ TESTS

'var()' functions are substituted at computed-value time. If a declaration, once all 'var()' functions are substituted in, does not match its declared grammar, the declaration is invalid at computed-value time.

▸ TESTS

If a declaration, once all 'var()' functions are substituted in, contains only a CSS-wide keyword (and possibly whitespace), its value is determined as if that keyword were its specified value all along.

▸ TESTS

EXAMPLE 13

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.

EXAMPLE 14

While a ‘var()’ function can’t get a CSS-wide keyword from the custom property itself—if you tried to specify that, like ‘--foo: initial;’, it would just trigger explicit defaulting for the custom property—it can have a CSS-wide keyword in its fallback:

```
p { color: var(--does-not-exist, initial); }
```

In the above code, if the ‘--does-not-exist’ property didn’t exist or is invalid at computed-value time, the ‘var()’ will instead substitute in the ‘initial’ keyword, making the property behave as if it was originally ‘color: initial’. This will make it take on the document’s initial ‘color’ value, rather than defaulting to inheritance, as it would if there were no fallback.

§ 3.1. Invalid Variables

When a custom property’s value is the guaranteed-invalid 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 the guaranteed-invalid 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 is one of the following depending on the property’s type:

↪ **The property is a non-registered custom property**
↪ **The property is a registered custom property with universal syntax**
The computed value is the guaranteed-invalid value.

↪ **Otherwise**
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.

▸ TESTS

EXAMPLE 15

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

'var()' functions produce some complications when parsing shorthand properties into their component longhands, and when serializing shorthand properties *from* their component longhands.

If a shorthand property contains a 'var()' function in its value, the longhand properties it's associated with must instead be filled in with a special, unobservable-to-authors ***pending-substitution value*** that indicates the shorthand contains a variable, and thus the longhand's value can't be determined until variables are substituted.

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.

▸ TESTS

Note: When a shorthand is written without a 'var()', it is parsed and separated out into its component longhand properties at parse time; the longhands then participate in the cascade, with the shorthand property more or less discarded. When the shorthand contains a 'var()', however, this can't be done, as the 'var()' could be substituted with anything.

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

▸ TESTS

✧ ✧ ✧

Shorthand properties are serialized by gathering the values of their component longhand properties, and synthesizing a value that will parse into the same set of values.

If all of the component longhand properties for a given shorthand are pending-substitution values from the same original shorthand value, the shorthand property must serialize to that original ('var()'-containing) value.

Otherwise, if any of the component longhand properties for a given shorthand are pending-substitution values, or contain 'var()' functions of their own that have not yet been substituted, the shorthand property must serialize to the empty string.

§ 3.3. Safely Handling Overly-Long Variables

Naively implemented, 'var()' functions can be used in a variation of the "billion laughs attack":

EXAMPLE 16

```
.foo {
  --prop1: lol;
  --prop2: var(--prop1) var(--prop1);
  --prop3: var(--prop2) var(--prop2);
  --prop4: var(--prop3) var(--prop3);
  /* etc */
}
```

In this short example, '--prop4''s computed value is 'lol lol lol lol lol lol lol lol', containing 8 copies of the original 'lol'. Every additional level added to this doubles the number of identifiers; extending it to a mere 30 levels, the work of a few minutes by hand, would make '--prop30' contain *nearly a billion instances* of the identifier.

To avoid this sort of attack, UAs must impose a UA-defined limit on the allowed length of the token stream that a 'var()' function expands into. If a 'var()' would expand into a longer token stream than this limit, it instead makes the property it's expanding into invalid at computed-value time.

▸ TESTS

This specification does not define what size limit should be imposed. However, since there are valid use-cases for custom properties that contain a kilobyte or more of text, it's recommended that the limit be set relatively high.

> Note: The general principle that UAs are allowed to violate standards due to resource constraints is still generally true here; a UA might, separately, have limits on how long of a custom property they can support, or how large of an identifier they can support. This section calls out this attack specifically because of its long history, and the fact that it can be done without any of the pieces *seeming* to be too large on first inspection.

§ 4. APIs

All custom property declarations have the case-sensitive flag set.

▸ TESTS

> 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 as the exact code point sequence provided by the author, including not altering the case.

> Note: For non-custom properties, property names are restricted to the ASCII range and are ASCII case-insensitive, so implementations typically serialize the name lowercased.

Specified values of custom properties must be serialized *exactly as specified by the author*. Simplifications that might occur in other properties, such as dropping comments, normalizing whitespace, reserializing numeric tokens from their value, etc., must not occur.

Computed values of custom properties must similarly be serialized *exactly as specified by the author*, save for the replacement of any 'var()' functions.

▸ TESTS

EXAMPLE 17

For example, given the following properties:

```
--y: /* baz */;
--x: /* foo */ var(--y) /* bar */;
```

the serialization of the specified value of '--x' must be `"/* foo */ var(--y) /* bar */"`, while the serialization of the computed value of '--x' must be `"/* foo */ /* baz */ /* bar */"`.

(Note that the leading whitespace on the value is automatically trimmed by the CSS parser; it's not preserved here.)

This requirement exists because authors sometimes store non-CSS information in custom properties, and "normalizing" this information can change it in ways that break author code. For example, storing a UUID in a custom property, like '--uuid: 12345678-12e3-8d9b-a456-426614174000', requires the UUID to be echoed back out as written when it's accessed by script.

This value is technically parsed by CSS as a series of adjacent numbers and dimensions. In particular, the segment "-12e3" parses as a number, equal to -12000. Reserializing it in that form, as required by CSSOM in other contexts, would fatally break the author's use of the value.

§ 5. Changes

§ 5.1. Changes Since the 11 November 2021 CR Draft

- Clarified that custom properties apply all pseudo-elements (including those with restricted property lists)
- Added example to illustrate issues with combining characters, ligatures, etc
- Strengthened wording around similar-appearing variable names that use distinct codepoint sequences
- Clarified an example by using more visualy distinct languages as examples (English and Greek)
- Split Security and Privacy into separate sections

§ 5.2. Changes Since the 03 December 2015 CR

- Now that [css-syntax-3] auto-trims whitespace from declaration values, made <declaration-value> optional in the custom property grammar, so that empty variables are still allowed. (Issue 774)
- Similarly, made empty fallbacks valid in 'var()'.
- The '-' property is reserved for future use by CSS.
- Added concept of "animation-tainted", to prevent non-animatable properties from using a variable to smuggle in some animatability.
- Defined the guaranteed-invalid value to make the initial value of custom properties and the result of cycles or substitution failure more straightforward, and allow failure to propagate thru substitutions until finally intercepted by a fallback.
- Defined that cycles trigger invalid at computed-value time behavior.
- Allowed variables to resolve to a CSS-wide keyword (only possible by providing it as a fallback).
- Clarified that registered custom properties act like non-custom properties when they're invalid at computed-value time.
- Made longhands with 'var()'s also trigger their shorthands to be unserializable, like longhands with pending-substitution values already did.
- Required UAs to defend against exponential substitution attacks.
- Defined how to serialize the *values* of custom properties (previously, only the property name's serialization was specified).

§ 5.3. 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.

7. Privacy Considerations

This specification defines a purely author-level mechanism for passing styling information around within a page they control. As such, there are no new privacy considerations.

8. Security Considerations

§ 3.3 Safely Handling Overly-Long Variables calls out a long-standing Denial-of-Service attack that can be mounted against "macro-expansion"-like mechanisms, such as the 'var()' function, and mandates a defense against that attack.

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 18
>
> This is an example of an informative example.

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

> Note, this is an informative note.

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

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

▼ TESTS

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

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
A CSS style sheet.

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

authoring tool
A UA that writes a style sheet.

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

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

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

§ Partial implementations

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

§ Implementations of Unstable and Proprietary Features

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

§ Non-experimental implementations

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

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

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

§ 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

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

- @keyframes

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

- background
- background-color

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

- margin-top

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

- all
- initial
- longhand property
- shorthand
- shorthand property
- specified value
- unset

[css-cascade-6] defines the following terms:

- cascade

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

- color
- transparent

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

- @media

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

- font-size

[css-properties-values-api-1] defines the following terms:

- registered custom property
- universal syntax definition

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

- <)-token>
- <]-token>
- <bad-string-token>
- <bad-url-token>
- <declaration-value>
- <delim-token>
- <semicolon-token>
- <}-token>

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

- ,
- <dashed-ident>
- ?
- calc()
- css-wide keywords
- em
- identifier

[cssom-1] defines the following terms:

- case-sensitive flag
- declarations
- getPropertyValue(property)

[INFRA] defines the following terms:

- ascii case-insensitive
- identical to

[selectors-4] defines the following terms:

- :lang()

[web-animations-1] defines the following terms:

- not animatable

§ References

§ Normative References

[CSS-ANIMATIONS-1]
Dean Jackson; et al. *CSS Animations Level 1*. 11 October 2018. WD. URL: https://www.w3.org/TR/css-animations-1/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. 13 January 2022. CR. URL: https://www.w3.org/TR/css-cascade-5/

[CSS-CONDITIONAL-3]
David Baron; Elika Etemad; Chris Lilley. *CSS Conditional Rules Module Level 3*. 13 January 2022. CR. URL: https://www.w3.org/TR/css-conditional-3/

[CSS-PROPERTIES-VALUES-API-1]
Tab Atkins Jr.; et al. *CSS Properties and Values API Level 1*. 13 October 2020. WD. URL: https://www.w3.org/TR/css-properties-values-api-1/

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

[CSS-VALUES-3]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. 6 June 2019. CR. URL: https://www.w3.org/TR/css-values-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/

[CSS2]
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/

[CSSOM-1]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. 26 August 2021. WD. URL: https://www.w3.org/TR/cssom-1/

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

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

[SELECTORS-4]
Elika Etemad; Tab Atkins Jr.. *Selectors Level 4*. 21 November 2018. WD. URL: https://www.w3.org/TR/selectors-4/

[WEB-ANIMATIONS-1]
Brian Birtles; et al. *Web Animations*. 18 May 2021. WD. URL: https://www.w3.org/TR/web-animations-1/

§ Informative References

[CHARMOD-NORM]
Addison Phillips; et al. *Character Model for the World Wide Web: String Matching*. 11 August 2021. NOTE. URL: https://www.w3.org/TR/charmod-norm/

[CSS-BACKGROUNDS-3]
Bert Bos; Elika Etemad; Brad Kemper. *CSS Backgrounds and Borders Module Level 3*. 26 July 2021. CR. URL: https://www.w3.org/TR/css-backgrounds-3/

[CSS-BOX-4]
Elika Etemad. *CSS Box Model Module Level 4*. 21 April 2020. WD. URL: https://www.w3.org/TR/css-box-4/

[CSS-CASCADE-6]

Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 6*. 21 December 2021. WD. URL: https://www.w3.org/TR/css-cascade-6/

[CSS-COLOR-4]

Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. 15 December 2021. WD. URL: https://www.w3.org/TR/css-color-4/

[CSS-EXTENSIONS]

Tab Atkins Jr.. *CSS Extensions*. ED. URL: https://drafts.csswg.org/css-extensions/

[CSS-FONTS-4]

John Daggett; Myles Maxfield; Chris Lilley. *CSS Fonts Module Level 4*. 21 December 2021. WD. URL: https://www.w3.org/TR/css-fonts-4/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Anim-ation type	Canonical order	Computed value
'--*'	<declaration-value>?	the guaranteed-invalid value	all elements and all pseudo-elements (including those with restricted property lists)	yes	n/a	discrete	per grammar	specified value with variables substituted, or the guaranteed-invalid value