CSS Values and Units Module Level 4

W3C Working Draft, 19 October 2022

▶ **More details about this document**

Abstract

This CSS module describes the common values and units that CSS properties accept and the syntax used for describing them in CSS property definitions.

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

Status of this document

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

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

This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code “css-values” in the title, like this: “[css-values] …*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

The value definition field of each CSS property can contain keywords, data types (which appear between '<' and '>'), and information on how they can be combined. Generic data types (<length> being the most widely used) that can be used by many properties are described in this specification, while more specific data types (e.g., <spacing-limit>) are described in the corresponding modules.

§ 1.1. Module Interactions

This module replaces and extends the data type definitions in [CSS21] sections 1.4.2.1, 4.3, and A.2.

§ 2. Value Definition Syntax

The ***value definition syntax*** described here is used to define the set of valid values for CSS properties (and the valid syntax of many other parts of CSS). A value so described can have one or more components.

§ 2.1. Component Value Types

Component value types are designated in several ways:

1. Keyword values (such as ‘auto’, ‘disc’, etc.), which appear literally, without quotes (e.g. `auto`).
2. Basic data types, which appear between ‘<’ and ‘>’ (e.g., <length>, <percentage>, etc.). For numeric data types, this type notation can annotate any range restrictions using the bracketed range notation described below.
3. Property value ranges, which represent the same pattern of values as a property bearing the same name. These are written as the property name, surrounded by single quotes, between ‘<’ and ‘>’, e.g., <'border-width'>, <'background-attachment'>, etc.

   These types *do not* include CSS-wide keywords such as ‘inherit’, and also *do not* include any top-level comma-separated-list multiplier (i.e. if a property named ‘pairing’ is defined as ‘[<custom-ident> <integer>?]#’, then ‘<'pairing'>’ is equivalent to ‘[<custom-ident> <integer>?]’, not ‘[<custom-ident> <integer>?]#’).
4. Functional notations and their arguments. These are written as the function’s name, followed by an empty parentheses pair, between ‘<’ and ‘>’, e.g. <calc()>.
5. Other non-terminals. These are written as the name of the non-terminal between ‘<’ and ‘>’, as in <spacing-limit>. Notice the distinction between <border-width> and <'border-width'>: the latter represents the grammar of the ‘border-width’ property, the former requires an explicit expansion elsewhere. The definition of a non-terminal is typically located near its first appearance in the specification.

Some property value definitions also include the slash (/), the comma (,), and/or parentheses as literals. These represent their corresponding tokens. Other non-keyword literal characters that may appear in a component value, such as “+”, must be written enclosed in single quotes.

***Commas* specified in the grammar are implicitly omissible** in some circumstances, when used to separate optional terms in the grammar. Within a top-level list in a property or other CSS value, or a function’s argument list, a comma specified in the grammar must be omitted if:

- all items preceding the comma have been omitted
- all items following the comma have been omitted
- multiple commas would be adjacent (ignoring white space/comments), due to the items between the commas being omitted.

EXAMPLE 1

For example, if a function can accept three arguments in order, but all of them are optional, the grammar can be written like:

```
example( first? , second? , third? )
```

Given this grammar, writing ‘example(first, second, third)’ is valid, as is ‘example(first, second)’ or ‘example(first, third)’ or ‘example(second)’. However, ‘example(first, , third)’ is invalid, as one of those commas are no longer separating two options; similarly, ‘example(,second)’ and ‘example(first,)’ are invalid. ‘example(first second)’ is also invalid, as commas are still required to actually separate the options.

If commas were not implicitly omittable, the grammar would have to be much more complicated to properly express the ways that the arguments can be omitted, greatly obscuring the simplicity of the feature.

All CSS properties also accept the CSS-wide keyword values as the sole component of their property value. For readability these are not listed explicitly in the property value syntax definitions. For example, the full value definition of ‘border-color’ under CSS Cascading and Inheritance Level 3 is `<color>{1,4} | inherit | initial | unset` (even though it is listed as `<color>{1,4}`).

Note: This implies that, in general, combining these keywords with other component values in the same declaration results in an invalid declaration. For example, ‘background: url(corner.png) no-repeat, inherit;’ is invalid.

§ 2.2. Component Value Combinators

Component values can be arranged into property values as follows:

- Juxtaposing components means that all of them must occur, in the given order.
- A double ampersand (**&&**) separates two or more components, all of which must occur, in any order.
- A double bar (||) separates two or more options: one or more of them must occur, in any order.
- A bar (|) separates two or more alternatives: exactly one of them must occur.
- Brackets ([ ]) are for grouping.

Juxtaposition is stronger than the double ampersand, the double ampersand is stronger than the double bar, and the double bar is stronger than the bar. Thus, the following lines are equivalent:

```
  a b   |   c ||   d &&   e f
[ a b ] | [ c || [ d && [ e f ]]]
```

For reorderable combinators (||, &&), ordering of the grammar does not matter: components in the same grouping may be interleaved in any order. Thus, the following lines are equivalent:

```
a || b || c
b || a || c
```

§ 2.3. Component Value Multipliers

Every type, keyword, or bracketed group may be followed by one of the following modifiers:

- An asterisk (*) indicates that the preceding type, word, or group occurs zero or more times.
- A plus (+) indicates that the preceding type, word, or group occurs one or more times.
- A question mark (*?*) indicates that the preceding type, word, or group is optional (occurs zero or one times).
- A single number in curly braces (*{***A***}*) indicates that the preceding type, word, or group occurs *A* times.
- A comma-separated pair of numbers in curly braces (*{***A**,**B***}*) indicates that the preceding type, word, or group occurs at least *A* and at most *B* times. The *B* may be omitted ({*A*,}) to indicate that there must be at least *A* repetitions, with no upper bound on the number of repetitions.
- A hash mark (#) indicates that the preceding type, word, or group occurs one or more times, separated by comma tokens (which may optionally be surrounded by white space and/or comments). It may optionally be followed by the curly brace forms, above, to indicate precisely how many times the repetition occurs, like '<length>#{1,4}'.
- An exclamation point (*!*) after a group indicates that the group is required and must produce at least one value; even if the grammar of the items within the group would otherwise allow the entire contents to be omitted, at least one component value must not be omitted.

The '+' and '#' multipliers may be stacked as '+#'; similarly, the '#' and '?' multipliers may be stacked as '#?'. These stacks each represent the later multiplier applied to the result of the earlier

multiplier. (These same stacks can be represented using grouping, but in complex grammars this can push the number of brackets beyond readability.)

For repeated component values (indicated by '*', '+', or '#'), UAs must support at least 20 repetitions of the component. If a property value contains more than the supported number of repetitions, the declaration must be ignored as if it were invalid.

§ 2.4. Combinator and Multiplier Patterns

There are a small set of common ways to combine multiple independent component values in particular numbers and orders. In particular, it's common to want to express that, from a set of component value, the author must select zero or more, one or more, or all of them, and in either the order specified in the grammar or in any order.

All of these can be easily expressed using simple patterns of combinators and multipliers:

	in order	**any order**
zero or more	`A? B? C?`	`A? || B? || C?`
one or more	`[ A? B? C? ]!`	`A || B || C`
all	`A B C`	`A && B && C`

Note that all of the "any order" possibilities are expressed using combinators, while the "in order" possibilities are all variants on juxtaposition.

§ 2.5. Component Values and White Space

Unless otherwise specified, white space and/or comments may appear before, after, and/or between components combined using the above combinators and multipliers.

> Note: In many cases, spaces will in fact be *required* between components in order to distinguish them from each other. For example, the value '1em2em' would be parsed as a single <dimension-token> with the number '1' and the identifier 'em2em', which is an invalid unit. In this case, a space would be required before the '2' to get this parsed as the two lengths '1em' and '2em'.

§ 2.6. Property Value Examples

Below are some examples of properties with their corresponding value definition fields

EXAMPLE 2

Property	Value definition field	Example value
'orphans'	<integer>	'3'
'text-align'	left | right | center | justify	'center'
'padding-top'	<length> | <percentage>	'5%'
'outline-color'	<color> | invert	'#fefefe'
'text-decoration'	none | underline || overline || line-through || blink	'overline underline'
'font-family'	[<family-name> | <generic-family>]#	'"Gill Sans", Futura, sans-serif'
'border-width'	[<length> | thick | medium | thin]{1,4}	'2px medium 4px'
'box-shadow'	[inset? && <length>{2,4} && <color>?]# | none	'3px 3px rgba(50%, 50%, 50%, 50%), lemonchiffon 0 0 4px inset'

§ 3. Combining Values: Interpolation, Addition, and Accumulation

Some procedures, for example transitions and animations, ***combine*** two CSS property values. The following combining operations—on the two computed values V_a and V_B yielding the computed value V_{result}—are defined:

interpolation

Given two property values V_a and V_B, produces an intermediate value V_{result} at a distance of p along the interval between V_a and V_B such that $p = 0$ produces V_a and $p = 1$ produces V_B.

The range of p is $(-\infty, \infty)$ due to the effect of timing functions. As a result, this procedure must also define extrapolation behavior for p outside [0, 1].

addition

Given two property values V_a and V_B, returns the sum of the two properties, V_{result}. For addition that is not commutative (for example, matrix multiplication) V_a represents the first term of the operation and V_B represents the second.

> Note: While addition can often be expressed in terms of the same weighted sum function used to define interpolation, this is not always the case. For example, interpolation of transform matrices involves decomposing and interpolating the matrix components whilst addition relies on matrix multiplication.

accumulation

Given two property values V_a and V_B, returns the result, V_{result}, of combining the two operands such that V_B is treated as a *delta* from V_a. For accumulation that is not commutative (for example, accumulation of mismatched transform lists) V_a represents the first term of the operation and V_B represents the second.

> Note: For many types of animation such as numbers or lengths, accumulation is defined to be identical to addition.
>
> A common case where the definitions differ is for list-based types where addition may be defined as appending to a list whilst accumulation may be defined as component-based addition. For example, the filter list values 'blur(2)' and 'blur(3)', when added together would produce 'blur(2) blur(3)', but when accumulated would produce 'blur(5)'.

These operations are only defined on computed values. (As a result, it is not necessary to define, for example, how to add a <length> value of '15pt' with '5em' since such values will be resolved to their canonical unit before being passed to any of the above procedures.)

If a value type does not define a specific procedure for addition or is defined as ***not additive***, its addition operation is simply $V_{result} = V_a$.

If a value type does not define a specific procedure for accumulation, its accumulation operation is identical to addition.

§ 3.1. Representing Interpolated Values: the 'mix()' notation

Interpolation of two values can be represented by the ***'mix()'*** functional notation, whose syntax is defined as follows:

```
mix( <percentage> ';' <start-value> ';' <end-value> )
```

'<percentage>'
: Represents the interpolation point as progress from <start-value> to <end-value>.

'<start-value>'
: The value at the "start" (0%) of the interpolation range.

'<end-value>'
: The value at the "end" (100%) of the interpolation range.

> Note: This functional notation uses semicolons to separate arguments rather than the more typical comma because the values themselves can contain commas.

A 'mix()' notation is invalid if either its <start-value> or <end-value> is invalid if substituted in its place, if it is not the sole value of the property, or if the property using it is not animatable.

> EXAMPLE 3
> For example, the following declarations are invalid, and will be ignored:
>
> ```
> color: mix(90% ; #invalid ; #F00);
> background: url(ocean) mix(10% ; blue ; yellow);
> display: mix(0% ; inline ; block);
> ```

Progress values below '0%' and above '100%' are valid; they represent interpolation beyond the range represented by the start and end values.

§ 3.2. Range Checking

Interpolation can result in a value outside the valid range for a property, even if all of the inputs to interpolation are valid; this especially happens when *p* is outside the [0, 1] range, but some easing functions can cause this to occur even within that range. If the final result *after* interpolation, addition, and accumulation is out-of-range for the target context the value is being used in, it does not cause the declaration to be invalid. Instead, the value must be clamped to the range allowed in the target context, exactly the same as math functions (see § 10.12 Range Checking).

Note: Even if interpolation results in an out-of-range value, addition/accumulation might "correct" the result and bring it back into range. Thus, clamping is only applied to the *final* result of applying all interpolation-related operations.

§ 4. Textual Data Types

The ***textual data types*** include various keywords and identifiers as well as strings (<string>) and URLs (<url>). Aside from the casing of pre-defined keywords or as explicitly defined for a given property, no normalization is performed, not even Unicode normalization: the specified and computed value of a property are exactly the provided Unicode values after parsing (which includes character set conversion and escaping). [UNICODE] [CSS-SYNTAX-3]

CSS ***identifiers***, generically denoted by ***'<ident>'***, consist of a sequence of characters conforming to the <ident-token> grammar. [CSS-SYNTAX-3] Identifiers cannot be quoted; otherwise they would be interpreted as strings. CSS properties accept two classes of identifiers: pre-defined keywords and author-defined identifiers.

Note: The <ident> production is not meant for property value definitions—<custom-ident> should be used instead. It is provided as a convenience for defining other syntactic constructs.

All textual data types interpolate as discrete and are not additive.

§ 4.1. Pre-defined Keywords

In the value definition fields, ***keywords*** with a pre-defined meaning appear literally. Keywords are identifiers and are interpreted ASCII case-insensitively (i.e., [a-z] and [A-Z] are equivalent).

EXAMPLE 4
For example, here is the value definition for the 'border-collapse' property:

```
Value: collapse | separate
```

And here is an example of its use:

```
table { border-collapse: separate }
```

§ 4.1.1. CSS-wide keywords: ‘initial’, ‘inherit’ and ‘unset’

As defined above, all properties accept the ***CSS-wide keywords***, which represent value computations common to all CSS properties. These keywords are normatively defined in the CSS Cascading and Inheritance Module.

Other CSS specifications can define additional CSS-wide keywords.

§ 4.2. Unprefixed Author-defined Identifiers: the <custom-ident> type

Some properties accept arbitrary author-defined identifiers as a component value. This generic data type is denoted by ***‘<custom-ident>’***, and represents any valid CSS identifier that would not be misinterpreted as a pre-defined keyword in that property’s value definition. Such identifiers are fully case-sensitive (meaning they’re compared using the "identical to" operation), even in the ASCII range (e.g. ‘example’ and ‘EXAMPLE’ are two different, unrelated user-defined identifiers).

The CSS-wide keywords are not valid <custom-ident>s. The ‘default’ keyword is reserved and is also not a valid <custom-ident>. Specifications using <custom-ident> must specify clearly what other keywords are excluded from <custom-ident>, if any—for example by saying that any pre-defined keywords in that property’s value definition are excluded. Excluded keywords are excluded in all ASCII case permutations.

When parsing positionally-ambiguous keywords in a property value, a <custom-ident> production can only claim the keyword if no other unfulfilled production can claim it.

> EXAMPLE 5
> For example, the shorthand declaration ‘animation: ease-in ease-out’ is equivalent to the longhand declarations ‘animation-timing-function: ease-in; animation-name: ease-out;’. ‘ease-in’ is claimed by the <easing-function> production belonging to ‘animation-timing-function’, leaving ‘ease-out’ to be claimed by the <custom-ident> production belonging to ‘animation-name’.

> Note: When designing grammars with <custom-ident>, the <custom-ident> should always be “positionally unambiguous”, so that it’s impossible to conflict with any keyword values in the property. Such conflicts can alternatively be avoided by using <dashed-ident>.

§ 4.3. Prefixed Author-defined Identifiers: the <dashed-ident> type

Some contexts accept *both* author-defined identifiers *and* CSS-defined identifiers. If not handled carefully, this can result in difficulties adding new CSS-defined values; UAs have to study existing usage and gamble that there are sufficiently few author-defined identifiers in use matching the new CSS-defined one, so giving the new value a special CSS-defined meaning won't break existing pages.

While there are many legacy cases in CSS that mix these two values spaces in exactly this fraught way, the <dashed-ident> type is meant to be an easy way to distinguish author-defined identifiers from CSS-defined identifiers.

The ***'<dashed-ident>'*** production is a <custom-ident>, with all the case-sensitivity that implies, with the additional restriction that it must start with two dashes (U+002D HYPHEN-MINUS).

<dashed-ident>s are reserved solely for use as author-defined names. CSS will never define a <dashed-ident> for its own use.

EXAMPLE 6

For example, custom properties need to be distinguishable from CSS-defined properties, as new properties are added to CSS regularly. To allow this, custom property names are required to be <dashed-ident>s, as in this example:

```
.foo {
  --fg-color: blue;
}
```

EXAMPLE 7

<dashed-ident>s are also used in the '@color-profile' rule, to separate author-defined color profiles from pre-defined ones like 'device-cmyk', and allow CSS to define more pre-defined (but overridable) profiles in the future without fear of clashing with author-defined profiles:

```
@color-profile --foo { src: url(https://example.com/foo.icc); }
.foo {
  color: color(--foo 1 0 .5 / .2);
}
```

EXAMPLE 8

CSS will use <dashed-ident> more in the future, as more author-controlled syntax is added. CSS authoring tools, such as preprocessors that turn custom syntax into standard CSS, *should* use <dashed-ident> as well, to avoid clashing with future CSS design.

For example, if a CSS preprocessor added a new "custom" at-rule, it *shouldn't* spell it '@custom', as this would clash with a future official '@custom' rule added by CSS. Instead, it should use '@--custom', which is guaranteed to never clash with anything defined by CSS.

Even better, it should use '@--library1-custom', so that if Library2 adds their own "custom" at-rule (spelled @--library2-custom), there's no possibility of clash. Ideally this prefix should be customizable, if allowed by the tooling, so authors can manually avoid clashes on their own.

§ 4.4. Quoted Strings: the <string> type

Strings are denoted by ***'<string>'***. When written literally, they consist of a sequence of characters delimited by double quotes or single quotes, corresponding to the <string-token> production in the CSS Syntax Module [CSS-SYNTAX-3].

EXAMPLE 9

Double quotes cannot occur inside double quotes, unless escaped (as `"\""` or as `"\22"`). Analogously for single quotes (`'\''` or `'\27'`).

```
content: "this is a 'string'.";
content: "this is a \"string\".";
content: 'this is a "string".';
content: 'this is a \'string\'.'
```

It is possible to break strings over several lines, for aesthetic or other reasons, but in such a case the newline itself has to be escaped with a backslash (\). The newline is subsequently removed from the string. For instance, the following two selectors are exactly the same:

EXAMPLE 10

```
a[title="a not s\
o very long title"] {/*...*/}
a[title="a not so very long title"] {/*...*/}
```

Since a string cannot directly represent a newline, to include a newline in a string, use the escape "\A". (Hexadecimal A is the line feed character in Unicode (U+000A), but represents the generic notion of "newline" in CSS.)

§ 4.5. Resource Locators: the <url> type

The <url> type, written with the ***'url()'*** and ***'src()'*** functions, represents a URL, which is a pointer to a resource.

The syntax of <url> is:

```
<url> = url( <string> <url-modifier>* ) |
        src( <string> <url-modifier>* )
```

EXAMPLE 11

This example shows a URL being used as a background image:

```
body { background: url("http://www.example.com/pinkish.gif") }
```

For legacy reasons, a 'url()' can be written without quotation marks around the URL itself, in which case it is specially-parsed as a <url-token> [CSS-SYNTAX-3]. Because of this special parsing, 'url()' is only able to specify its URL literally; 'src()' lacks this special parsing rule, and so its URL can be provided by functions, such as 'var()'.

EXAMPLE 12

For example, the following declarations are identical:

```
background: url("http://www.example.com/pinkish.gif");
background: url(http://www.example.com/pinkish.gif);
```

And these have the same meaning as well:

```
background: src("http://www.example.com/pinkish.gif");
--foo: "http://www.example.com/pinkish.gif";
background: src(var(--foo));
```

But this does *not* work:

```
--foo: "http://www.example.com/pinkish.gif";
background: url(var(--foo));
```

...because the unescaped "(" in the value causes a parse error, so the entire declaration is thrown out as invalid.

The unquoted ‘url()’ syntax cannot accept a <url-modifier> argument and has extra escaping requirements: parentheses, whitespace characters, single quotes (') and double quotes (") appearing in a URL must be escaped with a backslash, e.g. ‘url(open\(parens)’, ‘url(close\)parens)’. (In quoted <string> ‘url()’s, only newlines and the character used to quote the string need to be escaped.) Depending on the type of URL, it might also be possible to write these characters as URL-escapes (e.g. ‘url(open%28parens)’ or ‘url(close%29parens)’) as described in [URL].

The precise requirements for parsing the unquoted ‘url()’ syntax are normatively defined in [CSS-SYNTAX-3].

Some CSS contexts (such as ‘@import’) also allow a <url> to be represented by a bare <string>, without the function wrapper. In such cases the string behaves identically to a ‘url()’ function containing that string.

EXAMPLE 13

For example, the following statements act identically:

```
@import url("base-theme.css");
@import "base-theme.css";
```

4.5.1. Relative URLs

In order to create modular style sheets that are not dependent on the absolute location of a resource, authors should use relative URLs. Relative URLs (as defined in [URL]) are resolved to full URLs using a base URL. RFC 3986, section 3, defines the normative algorithm for this process. For CSS style sheets, the base URL is that of the style sheet itself, not that of the styled source document. Style sheets embedded within a document have the base URL associated with their container.

Note: For HTML documents, the base URL is mutable.

When a <url> appears in the computed value of a property, it is resolved to an absolute URL, as described in the preceding paragraph. The computed value of a URL that the UA cannot resolve to an absolute URL is the specified value.

EXAMPLE 14

For example, suppose the following rule:

```
body { background: url("tile.png") }
```

is located in a style sheet designated by the URL:

```
http://www.example.org/style/basic.css
```

The background of the source document's `<body>` will be tiled with whatever image is described by the resource designated by the URL:

```
http://www.example.org/style/tile.png
```

The same image will be used regardless of the URL of the source document containing the `<body>`.

§ *4.5.1.1. Fragment URLs*

To work around some common eccentricities in browser URL handling, CSS has special behavior for fragment-only urls.

If a <url>'s value starts with a U+0023 NUMBER SIGN (#) character, parse it as per normal for URLs, but additionally set the ***local url flag*** of the <url>.

When matching a <url> with the local url flag set, ignore everything but the URL's fragment, and resolve that fragment against the node tree of the stylesheet's owner node. This reference must always be treated as same-document (rather than cross-document).

ISSUE 1 This relative URL resolution behavior is under discussion. [Issue #3320]

Note: This means that such fragments will resolve against the contents of the current document (and in consideration of shadow DOM, only within the stylesheet's associated node tree), regardless of what base URL relative URLs outside of CSS resolve against.

EXAMPLE 15

In the following example, `#anchor` will resolve against `http://example.com/` whereas `#image` will resolve against the elements in the HTML document itself:

```
<!DOCTYPE html>
<base href="http://example.com/">
...
<a href="#anchor" style="background-image: url(#image)">link</a>
```

When serializing a ‘url()’ with the local url flag set, it must serialize as just the fragment.

▶ **What “browser eccentricities”?**

§ 4.5.2. Empty URLs

If the value of the <url> is the empty string (like ‘url("")’ or ‘url()’), the url must resolve to an invalid resource (similar to what the url ‘about:invalid’ does).

Its computed value is ‘url("")’ or ‘src("")’, whichever was specified, and it must serialize as such.

Note: This matches the behavior of empty urls for embedded resources elsewhere in the web platform, and avoids excess traffic re-requesting the stylesheet or host document due to editing mistakes leaving the ‘url()’ value empty, which are almost certain to be invalid resources for whatever the ‘url()’ shows up in. Linking on the web platform *does* allow empty urls, so if/when CSS gains some functionality to control hyperlinks, this restriction can be relaxed in those contexts.

§ 4.5.3. URL Modifiers

<url>s support specifying additional ***‘<url-modifier>’***s, which change the meaning or the interpretation of the URL somehow. A <url-modifier> is either an <ident> or a functional notation.

This specification does not define any <url-modifier>s, but other specs may do so.

Note: A <url> that is either unquoted or not wrapped in ‘url()’ notation cannot accept any <url-modifier>s.

§ 4.5.4. URL Processing Model

To ***fetch a style resource*** from url *url*, given a `CSSStyleSheet` *sheet*, a string *destination* matching a `RequestDestination`, a "no-cors" or "cors" *corsMode*, and an algorithm *processResponse* accepting a response and a null, failure or byte stream:

1. Let *environmentSettings* be *sheet*'s relevant settings object.
2. Let *documentBase* be *environmentSettings*'s API base URL.
3. Let *base* be *sheet*'s stylesheet base URL. [CSSOM]
4. Let *referrer* be *documentBase*.
5. If *base* is null, set *base* to *documentBase*.
6. Let *parsedUrl* be the result of the URL parser steps with *url* and *base*. If the algorithm returns an error, return.
7. If *corsMode* is "cors", set *referrer* to *sheet*'s location. [CSSOM]
8. Let *req* be a new request whose url is *parsedUrl*, whose destination is *destination*, mode is *corsMode*, origin is *environmentSettings*'s origin, credentials mode is "same-origin", use-url-credentials flag is set, client is *environmentSettings*, and whose referrer is *referrer*.
9. If *sheet*'s origin-clean flag is set, set *req*'s initiator type to "css". [CSSOM]
10. fetching *req*, with processresponseconsumebody set to *processResponse*.

> Note: Resources loaded through CSS style sheets are cached and cleared the same as any other resources linked from the document.

§ 5. Numeric Data Types

Numeric data types are used to represent quantities, indexes, positions, and other such values. Although many syntactic variations can exist in expressing the quantity (numeric aspect) in a given numeric value, the specified and computed value do not distinguish these variations: they represent the value's abstract quantity, not its syntactic representation.

The ***numeric data types*** include <integer>, <number>, <percentage>, and various dimensions including <length>, <angle>, <time>, <frequency>, and <resolution>.

> Note: While general-purpose dimensions are defined here, some other modules define additional data types (e.g. [css-grid-1] introduces 'fr' units) whose usage is more localized.

The precision and supported range of numeric values in CSS is *explicitly undefined*, and can vary based on the property or other context a value is used in. However, within the CSS specifications, infinite precision and range is assumed. When a value cannot be explicitly supported due to range/precision limitations, it must be converted to the closest value supported by the implementation, but how the implementation defines "closest" is explicitly undefined as well.

If an <angle> must be converted due to exceeding the implementation-defined range of supported values, it must be clamped to the nearest supported multiple of '360deg'.

§ 5.1. Range Restrictions and Range Definition Notation

Properties can restrict numeric values to some range. If the value is outside the allowed range, then unless otherwise specified, the declaration is invalid and must be ignored. Range restrictions can be annotated in the numeric type notation using ***CSS bracketed range notation***—`[min,max]`—within the angle brackets, after the identifying keyword, indicating a closed range between (and including) *min* and *max*. For example, <integer [0,10]> indicates an integer between '0' and '10', inclusive, while <angle [0,180deg]> indicates an angle between '0deg' and '180deg' (expressed in any unit).

> Note: CSS values generally do not allow open ranges; thus only square-bracket notation is used.

CSS theoretically supports infinite precision and infinite ranges for all value types; however in reality implementations have finite capacity. UAs should support reasonably useful ranges and precisions. Range extremes that are ideally unlimited are indicated using ∞ or −∞ as appropriate. For example, <length [0,∞]> indicates a non-negative length.

If no range is indicated, either by using the bracketed range notation or in the property description, then `[-∞,∞]` is assumed.

Values of −∞ or ∞ must be written without units, even if the value type uses units. Values of '0' *can* be written without units, even if the value type doesn't allow "unitless zeroes" (such as <time>).

> Note: At the time of writing, the bracketed range notation is new; thus in most CSS specifications any range limitations are described only in prose. (For example, "Negative values are not allowed" or "Negative values are invalid" indicate a `[0,∞]` range.) This does not make them any less binding.

§ 5.2. Integers: the <integer> type

Integer values are denoted by ***‘<integer>’***.

When written literally, an ***integer*** is one or more decimal digits ‘0’ through ‘9’ and corresponds to a subset of the <number-token> production in the CSS Syntax Module [CSS-SYNTAX-3]. The first digit of an integer may be immediately preceded by ‘-’ or ‘+’ to indicate the integer’s sign.

Unless otherwise specified, in the CSS specifications ***rounding to the nearest integer*** requires rounding in the direction of +∞ when the fractional portion is exactly 0.5. (For example, ‘1.5’ rounds to ‘2’, while ‘-1.5’ rounds to ‘-1’.)

§ 5.2.1. Computation and Combination of <integer>

Unless otherwise specified, the computed value of a specified <integer> is the specified abstract integer.

Interpolation of <integer> is defined as $V_{\text{result}} = \text{round}((1 - p) \times V_a + p \times V_b)$; that is, interpolation happens in the real number space as for <number>s, and the result is converted to an <integer> by rounding to the nearest integer.

Addition of <integer> is defined as $V_{result} = V_a + V_b$

§ 5.3. Real Numbers: the <number> type

Number values are denoted by ***‘<number>’***, and represent real numbers, possibly with a fractional component.

When written literally, a ***number*** is either an integer, or zero or more decimal digits followed by a dot (.) followed by one or more decimal digits; optionally, it can be concluded by the letter “e” or “E” followed by an integer indicating the base-ten exponent in scientific notation. It corresponds to the <number-token> production in the CSS Syntax Module [CSS-SYNTAX-3]. As with integers, the first character of a number may be immediately preceded by ‘-’ or ‘+’ to indicate the number’s sign.

The value ***‘<zero>’*** represents a literal number with the value 0. Expressions that merely evaluate to a <number> with the value 0 (for example, ‘calc(0)’) do not match <zero>; only literal <number-token>s do.

§ 5.3.1. Computation and Combination of <number>

Unless otherwise specified, the computed value of a specified <number> is the specified abstract number.

Interpolation of <number> is defined as $V_{\text{result}} = (1 - p) \times V_a + p \times V_b$

Addition of <number> is defined as $V_{result} = V_a + V_b$

§ 5.4. Numbers with Units: dimension values

The general term ***dimension*** refers to a number with a unit attached to it; and is denoted by ***'<dimension>'***.

When written literally, a dimension is a number immediately followed by a unit identifier, which is an identifier. It corresponds to the <dimension-token> production in the CSS Syntax Module [CSS-SYNTAX-3]. Like keywords, unit identifiers are ASCII case-insensitive.

CSS uses <dimension>s to specify distances (<length>), durations (<time>), frequencies (<frequency>), resolutions (<resolution>), and other quantities.

§ 5.4.1. Compatible Units

When serializing computed values [CSSOM], ***compatible units*** (those related by a static multiplicative factor, like the 96:1 factor between 'px' and 'in', or the computed 'font-size' factor between 'em' and 'px') are converted into a single ***canonical unit***. Each group of compatible units defines which among them is the canonical unit that will be used for serialization.

When serializing resolved values that are used values, all value types (percentages, numbers, keywords, etc.) that represent lengths are considered compatible with lengths. Likewise any future API that returns used values must consider any values that represent distances/durations/frequencies/etc. as compatible with the relevant class of dimensions, and canonicalize accordingly.

§ 5.4.2. Combination of Dimensions

Interpolation of compatible dimensions (for example, two <length> values) is defined as $V_{\text{result}} = (1 - p) \times V_a + p \times V_b$

Addition of compatible dimensions is defined as $V_{result} = V_a + V_b$

§ 5.5. Percentages: the <percentage> type

Percentage values are denoted by ***'<percentage>'***, and indicates a value that is some fraction of another reference value.

When written literally, a ***percentage*** consists of a number immediately followed by a percent sign '%'. It corresponds to the <percentage-token> production in the CSS Syntax Module [CSS-SYNTAX-3].

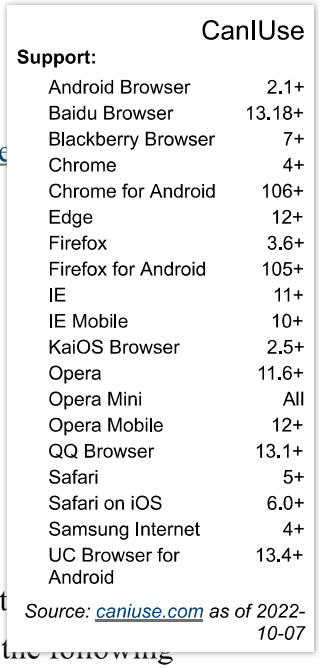
Percentage values are always relative to another quantity, for example a length. Each property that allows percentages also defines the quantity to which the percentage refers. This quantity can be a value of another property for the same element, the value of a property for an ancestor element, a measurement of the formatting context (e.g., the width of a containing block), or something else.

§ 5.5.1. Computation and Combination of <percentage>

Unless otherwise specified (such as in 'font-size', which computes its <percentage> to a <length>), the computed value of a percentage is the specified percentage.

Interpolation of <percentage> is defined as $V_{\text{result}} = (1 - p) \times V_a + p \times V_b$

Addition of <percentage> is defined as $V_{result} = V_a + V_b$



§ 5.6. Mixing Percentages and Dimensions

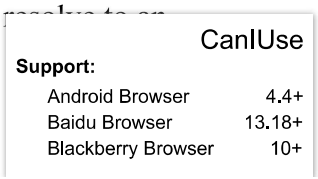
In cases where a <percentage> can represent the same quantity as a dimension in t value position, and can therefore be combined with them in a 'calc()' expression, the following convenience notations may be used in the property grammar:

'<length-percentage>'
: Equivalent to `[ <length> | <percentage> ]`, where the <percentage> will resolve to a <length>.

'<frequency-percentage>'
: Equivalent to `[ <frequency> | <percentage> ]`, where the <percentage> will resolve to a <frequency>.

'<angle-percentage>'
: Equivalent to `[ <angle> | <percentage> ]`, where the <percentage> will resolve to an <angle>.

'<time-percentage>'



Equivalent to [<time> | <percentage>], where the <percentage> will re

Chrome	27+
Chrome for Android	106+
Edge	12+
Firefox	2+
Firefox for Android	105+
IE (limited)	9+
IE Mobile	10+
KaiOS Browser	2.5+
Opera	15+
Opera Mini	None
Opera Mobile	64+
QQ Browser	13.1+
Safari	7+
Safari on iOS	7.0+
Samsung Internet	4+
UC Browser for Android	13.4+

Source: caniuse.com as of 2022-10-07

EXAMPLE 16

For example, the 'width' property can accept a <length> or a <percentage>, bot measure of distance. This means that 'width: calc(500px + 50%);' is allowed— converted to absolute lengths and added. If the containing block is '1000px' wi 50%;' is equivalent to 'width: 500px', and 'width: calc(50% + 500px)' thus end 'width: calc(500px + 500px)' or 'width: 1000px'.

On the other hand, the second and third arguments of the 'hsl()' function can on <percentage>s. Although 'calc()' productions are allowed in their place, they ca percentages with themselves, as in 'calc(10% + 20%)'.

Note: Specifications should never alternate <percentage> in place of a dimension in a grammar unless they are compatible.

Note: More <*type*-percentage> productions can be added in the future as needed. A <number-percentage> will never be added, as <number> and <percentage> can't be combined in 'calc()'.

§ 5.6.1. Computation and Combination of Percentage and Dimension Mixes

The computed value of a percentage-dimension mix is defined as

- a computed dimension if the percentage component is zero or is defined specifically to compute to a dimension value
- a computed percentage if the dimension component is zero
- a computed calc() expression otherwise

Interpolation of percentage-dimension value combinations (e.g. <length-percentage>, <frequency-percentage>, <angle-percentage>, <time-percentage> or equivalent notations) is defined as

- equivalent to interpolation of <length> if both V_a and V_b are pure <length> values
- equivalent to interpolation of <percentage> if both V_a and V_b are pure <percentage> values
- equivalent to converting both values into a 'calc()' expression representing the sum of the dimension type and a percentage (each possibly zero) and interpolating each component individually (as a <length>/<frequency>/<angle>/<time> and as a <percentage>, respectively)

Addition of <percentage> is defined the same as interpolation except by adding each component rather than interpolating it.

§ 5.7. Ratios: the <ratio> type

Ratio values are denoted by ***'<ratio>'***, and represent the ratio of two numeric values. It most often represents an aspect ratio, relating a width (first) to a height (second).

When written literally, a ***ratio*** has the syntax:

```
<ratio> = <number [0,∞]> [ / <number [0,∞]> ]?
```

The second <number> is optional, defaulting to '1'. However, <ratio> is always serialized with both components.

The computed value of a <ratio> is the pair of numbers provided.

If either number in the <ratio> is 0 or infinite, it represents a ***degenerate ratio*** (and, generally, won't do anything).

If two <ratio>s need to be compared, divide the first number by the second, and compare the results. For example, '3/2' is less than '2/1', because it resolves to 1.5 while the second resolves to 2. (In other words, "tall" aspect ratios are less than "wide" aspect ratios.)

§ 5.7.1. Combination of <ratio>

The interpolation of a <ratio> is defined by converting each <ratio> to a number by dividing the first value by the second (so a ratio of '3 / 2' would become '1.5'), taking the logarithm of that result (so the '1.5' would become approximately '0.176'), then interpolating those values. The result during the interpolation is converted back to a <ratio> by inverting the logarithm, then interpreting the result as a <ratio> with the result as the first value and '1' as the second value.

If either <ratio> is degenerate, the values cannot be interpolated.

EXAMPLE 17

For example, halfway through a linear interpolation from ‘5 / 1’ to ‘3 / 2’, the re
approximately the ratio ‘2.73 / 1’ (roughly ‘11 / 4’, slightly taller than a ‘3 / 1’ r

```
start  = log(5);   // ≈ 0.69897
end    = log(1.5); // ≈ 0.17609
interp = 0.69897*.5 + 0.17609*.5; // ≈ 0.43753
final  = 10^interp; // ≈ 2.73
```

IE (limited)	9+
IE Mobile (limited)	10+
KaiOS Browser	2.5+
Opera	15+
Opera Mini	None
Opera Mobile	64+
QQ Browser	13.1+
Safari	6.1+
Safari on iOS	8+
Samsung Internet	4+
UC Browser for Android	13.4+

Source: caniuse.com as of 2022-10-07

Note: Interpolating over the logarithm of the ratio means the results are scale-independent (‘5 / 1’ to ‘300 / 200’ would give the same results as above), that they’re symmetrical over "wide" and "tall" variants (interpolating from ‘1 / 5’ to ‘2 / 3’ would give a ratio approximately equal to ‘1 / 2.73’ at the halfway point), and that they’re symmetrical over whether the width is fixed and the height is based on the ratio or vice versa. These properties are not shared by many other possible interpolation strategies.

Note: Due to the properties of logarithms, any log can be used; the example here uses base-10 log, but if, say, the natural log and e was used, the intermediate results would be different but the final result would be the same.

Addition of <ratio>s is not possible.

§ 6. Distance Units: the <length> type

Lengths refer to distance measurements and are denoted by ***‘<length>’*** in the property definitions. A length is a dimension.

For zero lengths the unit identifier is optional (i.e. can be syntactically represented as the <number> ‘0’). However, if a ‘0’ could be parsed as either a <number> or a <length> in a property (such as ‘line-height’), it must parse as a <number>.

Properties may restrict the length value to some range. If the value is outside the allowed range, the declaration is invalid and must be ignored.

While some properties allow negative length values, this may complicate the formatting and there may be implementation-specific limits. If a negative length value is allowed but cannot be supported, it must be converted to the nearest value that can be supported.

In cases where the used length cannot be supported, user agents must approximate it in the actual value.

There are two types of length units: relative and absolute. The specified value of a length (***specified length***) is represented by its quantity and its unit. The computed value of a length (***computed length***) is the specified length resolved to an absolute length, and its unit is not distinguished: it can be represented by any absolute length unit (but will be serialized using its canonical unit, 'px').

§ 6.1. Relative Lengths

Relative length units specify a length relative to another length. Style sheets that use relative units can more easily scale from one output environment to another.

The relative units are:

Informative Summary of Relative Units

unit	relative to
'em'	font size of the element
'ex'	x-height of the element's font
'cap'	cap height (the nominal height of capital letters) of the element's font
'ch'	typical character advance of a narrow glyph in the element's font, as represented by the "0" (ZERO, U+0030) glyph
'ic'	typical character advance of a fullwidth glyph in the element's font, as represented by the "水" (CJK water ideograph, U+6C34) glyph
'rem'	font size of the root element
'lh'	line height of the element
'rlh'	line height of the root element
'vw'	1% of viewport's width
'vh'	1% of viewport's height
'vi'	1% of viewport's size in the root element's inline axis

unit	relative to
‘vb’	1% of viewport’s size in the root element’s block axis
‘vmin’	1% of viewport’s smaller dimension
‘vmax’	1% of viewport’s larger dimension

Child elements do not inherit the relative values as specified for their parent; they inherit the computed values.

§ 6.1.1. Font-relative Lengths: the ‘em’, ‘rem’, ‘ex’, ‘rex’, ‘cap’, ‘rcap’, ‘ch’, ‘rch’, ‘ic’, ‘ric’, ‘lh’, ‘rlh’ units

The ***font-relative lengths*** refer to the font metrics either of the element on which they are used (for the ***local font-relative lengths***) or of the root element (for the ***root font-relative lengths***).

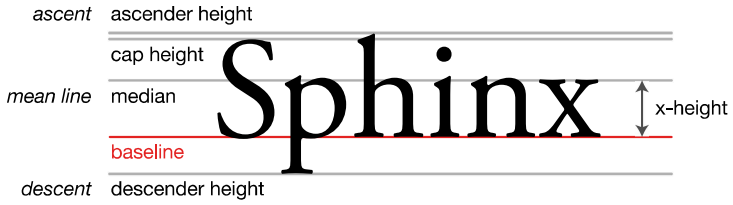


Figure 1 *Common typographic metrics*

‘em unit’

Equal to the computed value of the ‘font-size’ property of the element on which it is used.

EXAMPLE 18

The rule:

```
h1 { line-height: 1.2em }
```

means that the line height of `h1` elements will be 20% greater than the font size of `h1` element. On the other hand:

```
h1 { font-size: 1.2em }
```

means that the font size of `h1` elements will be 20% greater than the computed font size inherited by `h1` elements.

'rem unit'

Equal to the computed value of the 'em' unit on the root element.

'ex unit'

Equal to the used x-height of the first available font [CSS3-FONTS]. The x-height is so called because it is often equal to the height of the lowercase "x". However, an 'ex' is defined even for fonts that do not contain an "x". The x-height of a font can be found in different ways. Some fonts contain reliable metrics for the x-height. If reliable font metrics are not available, UAs may determine the x-height from the height of a lowercase glyph. One possible heuristic is to look at how far the glyph for the lowercase "o" extends below the baseline, and subtract that value from the top of its bounding box. In the cases where it is impossible or impractical to determine the x-height, a value of 0.5em must be assumed.

'rex unit'

Equal to the value of the 'ex' unit on the root element.

'cap unit'

Equal to the used cap-height of the first available font [CSS3-FONTS]. The cap-height is so called because it is approximately equal to the height of a capital Latin letter. However, a 'cap' is defined even for fonts that do not contain Latin letters. The cap-height of a font can be found in different ways. Some fonts contain reliable metrics for the cap-height. If reliable font metrics are not available, UAs may determine the cap-height from the height of an uppercase glyph. One possible heuristic is to look at how far the glyph for the uppercase "O" extends below the baseline, and subtract that value from the top of its bounding box. In the cases where it is impossible or impractical to determine the cap-height, the font's ascent must be used.

'rcap unit'

Equal to the value of the ‘cap’ unit on the root element.

‘ch unit’

Represents the typical advance measure of European alphanumeric characters, and measured as the used advance measure of the “0” (ZERO, U+0030) glyph in the font used to render it. (The ***advance measure*** of a glyph is its advance width or height, whichever is in the inline axis of the element.)

> Note: This measurement is an approximation (and in monospace fonts, an exact measure) of a single narrow glyph’s advance measure, thus allowing measurements based on an expected glyph count.

> Note: The advance measure of a glyph depends on writing-mode and text-orientation as well as font settings, text-transform, and any other properties that affect glyph selection or orientation.

In the cases where it is impossible or impractical to determine the measure of the “0” glyph, it must be assumed to be 0.5em wide by 1em tall. Thus, the ‘ch’ unit falls back to ‘0.5em’ in the general case, and to ‘1em’ when it would be typeset upright (i.e. ‘writing-mode’ is ‘vertical-rl’ or ‘vertical-lr’ and ‘text-orientation’ is ‘upright’).

‘rch unit’

Equal to the value of the ‘ch’ unit on the root element.

‘ic unit’

Represents the typical advance measure of CJK letters, and measured as the used advance measure of the “水” (CJK water ideograph, U+6C34) glyph found in the font used to render it.

> Note: This measurement is a typically an exact measure (in the few fonts with proportional fullwidth glyphs, an approximation) of a single fullwidth glyph’s advance measure, thus allowing measurements based on an expected glyph count.

In the cases where it is impossible or impractical to determine the ideographic advance measure, it must be assumed to be 1em.

‘ric unit’

Equal to the value of the ‘ic’ unit on the root element.

‘lh unit’

Equal to the computed value of the ‘line-height’ property of the element on which it is used, converting ‘normal’ to an absolute length by using only the metrics of the first available font.

'rlh unit'

Equal to the value of the 'lh' unit on the root element.

> Note: Setting the 'height' of an element using either the 'lh' or the 'rlh' units does not enable authors to control the actual number of lines in that element. These units only enable length calculations based on the theoretical size of an ideal empty line; the size of actual lines boxes may differ based on their content. In cases where an author wants to limit the number of actual lines in an element, the 'max-lines' property can be used instead.

When used in the value of the 'font-size' property on the element they refer to, the local font-relative lengths resolve against the computed metrics of the parent element—or against the computed metrics corresponding to the initial values of the 'font' and 'line-height' properties, if the element has no parent. Likewise, when 'lh' or 'rlh' units are used in the value of the 'line-height' property on the element they refer to, they resolve against the computed 'line-height' and font metrics of the parent element—or the computed metrics corresponding to the initial values of the 'font' and 'line-height' properties, if the element has no parent. (The other font-relative lengths continue to resolve against the element's own metrics when used in 'line-height'.)

When used outside the context of an element (such as in media queries), the font-relative lengths units refer to the metrics corresponding to the initial values of the 'font' and 'line-height' properties. Similarly, when specified in a document with no root element, the root font-relative lengths are resolved assuming the initial values of the 'font' and 'line-height' properties.

> Note: Font-relative units such as 'ch' and 'ic' can trigger font downloads, if a required font is not yet loaded.

The font-relative lengths are calculated in the absence of shaping.

Some user-agents allow users to apply additional restrictions to font sizes in a document, such as setting minimum font sizes to ensure readability. Such restrictions must be applied to the used value of the affected properties only; they *must not* affect the resolution of font-relative lengths used in properties. However, in other contexts (such as in media queries), to the extent that they would impact the used font metrics, such restrictions *do* affect the resolution of font-relative lengths.

> Note: In general, respecting a user's preferences, like minimum font sizes, is desirable; it's useful for a media query like '(min-width: 40em)' to use the actual font size the document will be displayed in. However, having these preferences affect font-relative lengths *in properties on an element* was found to not be Web-compatible; too many pages expect these units to be exact multiples of the specified 'font-size', rather than the *actual* font-size after applying user preferences.

Some user-agents apply restrictions to the 'line-height' values on form controls. These must have no effect on the 'lh' and 'rlh' units. The effect on their descendants, however, is undefined.

§ 6.1.2. Viewport-percentage Lengths: the '*vw', '*vh', '*vi', '*vb', '*vmin', '*vmax' units

The ***viewport-percentage lengths*** are relative to the size of the initial containing block—which is itself based on the size of either the viewport (for continuous media) or the page area (for paged media). When the height or width of the initial containing block is changed, they are scaled accordingly.

§ *6.1.2.1. The Large, Small, and Dynamic Viewport Sizes*

There are four variants of the viewport-percentage length units, corresponding to four (possibly identical) notions of the viewport size.

UA-default viewport

The ***UA-default viewport-percentage units*** ('v*') are defined with respect to a UA-defined ***UA-default viewport size***, which for any given document should be equivalent to the large viewport size, small viewport size, or some intermediary size.

> ISSUE 2 Should the UA-default viewport size be required to correspond to the size of the initial containing block?

> Note: Implementations that choose a size other than the large viewport size or small viewport size are encouraged to explain their choice to the CSSWG for consideration in future specification updates.

large viewport

The ***large viewport-percentage units*** ('lv*') are defined with respect to the ***large viewport size***: the viewport sized assuming any UA interfaces that are dynamically expanded and retracted to be

retracted. This allows authors to size content such that it is guaranteed to fill the viewport, noting that such content might be hidden behind such interfaces when they are expanded.

The sizes of the large viewport-percentage units are fixed (and therefore stable) unless the viewport itself is resized.

> EXAMPLE 19
> For example, on phones, where screen real-estate is at a premium, browsers will often hide part or all of the title and address bar once the user starts scrolling the page. The large viewport-percentage units are sized relative to this larger everything-retracted space, so content using these units will fill the entire visible page when these UI elements are hidden. However, when these retractable elements are shown, they can obscure content that is sized or positioned using these units.

small viewport

The ***small viewport-percentage units*** (‘sv*’) are defined with respect to the ***small viewport size***: the viewport sized assuming any UA interfaces that are dynamically expanded and retracted to be expanded. This allows authors to size content such that it can fit within the viewport even when such interfaces are present, noting that such content might not fill the viewport when such interfaces are retracted.

The sizes of the small viewport-percentage units are fixed (and therefore stable) unless the viewport itself is resized.

> EXAMPLE 20
> An element that is sized as ‘height: 100svh’, for example, will fill the screen perfectly, without any of its content being obscured, when all the dynamic UI elements of the UA are shown.
>
> Once those UI elements start being hidden, however, there will be extra space around the element. The small viewport-percentage units units are thus “safer” in general, but might not produce the most attractive layout once the user starts interacting with the page.

dynamic viewport

The ***dynamic viewport-percentage units*** (‘dv*’) are defined with respect to the ***dynamic viewport size***: the viewport sized with dynamic consideration of any UA interfaces that are dynamically expanded and retracted. This allows authors to size content such that it can exactly fit within the viewport whether or not such interfaces are present.

The sizes of the dynamic viewport-percentage units *are not stable* even while the viewport itself is unchanged. Using these units can cause content to resize e.g. while the user scrolls the page. Depending on usage, this can be disturbing to the user and/or costly in terms of performance.

The UA is not required to animate the dynamic viewport-percentage units while expanding and retracting any relevant interfaces, and may instead calculate the units as if the relevant interface was fully expanded or retracted during the UI animation. (It is recommended that UAs assume the fully-retracted size for this duration.)

Whether the expansion/retraction of a particular interface (A) changes the sizes of all of the viewport-percentage lengths (and the initial containing block) simultaneously or (B) contributes to the differences between the large viewport size and small viewport size is largely UA-dependent. However:

- Changes in interface that happen as a result of scrolling or other frequent page interactions that would disturb the user if they resulted in substantial layout changes must be categorized as the latter (B).
- Changes in interface that have a sufficiently steady state that re-laying out the document into the adjusted space would be beneficial to the user must be categorized as the former (A).
- Additionally, UAs may have some dynamically-shown interfaces that intentionally overlay content and do not cause any shifts in layout—and therefore have no effect on any of the viewport-percentage lengths. (Typically on-screen keyboards will fit into this category.)

In all cases, scrollbars are assumed not to exist. Note however that the initial containing block's size *is* affected by the presence of scrollbars on the viewport.

ISSUE 3 Level 3 assumes scrollbars never exist because it was hard to implement and only Firefox bothered to do so. This is making authors unhappy. Can we improve here?

§ *6.1.2.2. The Various Viewport-relative Units*

The viewport-percentage length units are:

'vw unit'*, *'svw unit'*, *'lvw unit'*, *'dvw unit'
: Equal to 1% of the width of the UA-default viewport size, small viewport size, large viewport size, and dynamic viewport size, respectively.

EXAMPLE 21

In the example below, if the width of the viewport is 200mm, the font size of `h1` elements will be 16mm (i.e. (8×200mm)/100).

```
h1 { font-size: 8vw }
```

'vh unit' , ***'svh unit'*** , ***'lvh unit'*** , ***'dvh unit'***

Equal to 1% of the height of the UA-default viewport size, small viewport size, large viewport size, and dynamic viewport size, respectively.

'vi unit' , ***'svi unit'*** , ***'lvi unit'*** , ***'dvi unit'***

Equal to 1% of the size of the large viewport size, small viewport size, and dynamic viewport size (respectively) in the box's inline axis.

'vb unit' , ***'svb unit'*** , ***'lvb unit'*** , ***'dvb unit'***

Equal to 1% of the size of the initial containing block UA-default viewport size, small viewport size, large viewport size, and dynamic viewport size (respectively) in the box's block axis.

'vmin unit' , ***'svmin unit'*** , ***'lvmin unit'*** , ***'dvmin unit'***

Equal to the smaller of '*vw' or '*vh'.

'vmax unit' , ***'svmax unit'*** , ***'lvmax unit'*** , ***'dvmax unit'***

Equal to the larger of '*vw' or '*vh'.

ISSUE 4 Originally the (unprefixed) viewport units were defined relative to the viewport size in general. The dynamism of browser chrome shifting in and out during scrolling was invented later, and following Safari's lead, most UAs mapped these units to the larger size. Defining it this way is prettier in many cases, but can also block critical content (such as toolbars, headers, and footers) in others. It's therefore not entirely clear whether this is the best mapping.

In situations where there is no element or it hasn't yet been styled (such as when evaluating media queries), the '*vi' and '*vb' units use the initial value of the 'writing-mode' property to determine which axis they correspond to.

§ 6.2. Absolute Lengths: the 'cm', 'mm', 'Q', 'in', 'pt', 'pc', 'px' units

The ***absolute length units*** are fixed in relation to each other and anchored to some physical measurement. They are mainly useful when the output environment is known. The absolute units consist of the ***physical units*** ('in', 'cm', 'mm', 'pt', 'pc', 'Q') and the ***visual angle unit (pixel unit)*** ('px'):

unit	name	equivalence
'cm'	centimeters	1cm = 96px/2.54
'mm'	millimeters	1mm = 1/10th of 1cm
'Q'	quarter-millimeters	1Q = 1/40th of 1cm
'in'	inches	1in = 2.54cm = 96px
'pc'	picas	1pc = 1/6th of 1in
'pt'	points	1pt = 1/72nd of 1in
'px'	pixels	1px = 1/96th of 1in

EXAMPLE 22

```
h1 { margin: 0.5in }      /* inches  */
h2 { line-height: 3cm }   /* centimeters */
h3 { word-spacing: 4mm }  /* millimeters */
h3 { letter-spacing: 1Q } /* quarter-millimeters */
h4 { font-size: 12pt }    /* points */
h4 { font-size: 1pc }     /* picas */
p  { font-size: 12px }    /* px */
```

Note: Lengths in publishing contexts are sometimes written like `2p3`, indicating a length of 2 picas and 3 points. These can be written in CSS as 'calc(2pc + 3pt)' (see § 10.1 Basic Arithmetic: calc()).

All of the absolute length units are compatible, and 'px' is their canonical unit.

For a CSS device, these dimensions are ***anchored*** either

i. by relating the physical units to their physical measurements, or

ii. by relating the pixel unit to the reference pixel.

For print media at typical viewing distances, the anchor unit should be one of the physical units (inches, centimeters, etc). For screen media (including high-resolution devices), low-resolution

devices, and devices with unusual viewing distances, it is recommended instead that the anchor unit be the pixel unit. For such devices it is recommended that the pixel unit refer to the whole number of device pixels that best approximates the reference pixel.

> Note: If the anchor unit is the pixel unit, the physical units might not match their physical measurements. Alternatively if the anchor unit is a physical unit, the pixel unit might not map to a whole number of device pixels.

> Note: This definition of the pixel unit and the physical units differs from the earlier editions of CSS1 and CSS2. In particular, in previous versions of CSS the pixel unit and th
> were not related by a fixed ratio: the physical units were always tied to their phy
> measurements while the pixel unit would vary to most closely match the referer
> unfortunate change was made because too much existing content relies on the a
> and breaking that assumption broke the content.)

> Note: Units are ASCII case-insensitive and serialize as lower case, for example

The ***reference pixel*** is the visual angle of one pixel on a device with a device pixe
and a distance from the reader of an arm's length. For a nominal arm's length of 2
angle is therefore about 0.0213 degrees. For reading at arm's length, 1px thus corr
0.26 mm (1/96 inch).

The image below illustrates the effect of viewing distance on the size of a reference pixel: a reading distance of 71 cm (28 inches) results in a reference pixel of 0.26 mm, while a reading distance of 3.5 m (12 feet) results in a reference pixel of 1.3 mm.

CanIUse

Support:

Android Browser	106+
Baidu Browser	13.18+
Blackberry Browser	10+
Chrome	26+
Chrome for Android	106+
Edge	12+
Firefox	16+
Firefox for Android	105+
IE (limited)	9+
IE Mobile	10+
KaiOS Browser	2.5+
Opera	15+
Opera Mini	None
Opera Mobile	64+
QQ Browser	13.1+
Safari	6.1+
Safari on iOS	7.0+
Samsung Internet	4+
UC Browser for Android	13.4+

Source: caniuse.com as of 2022-10-07

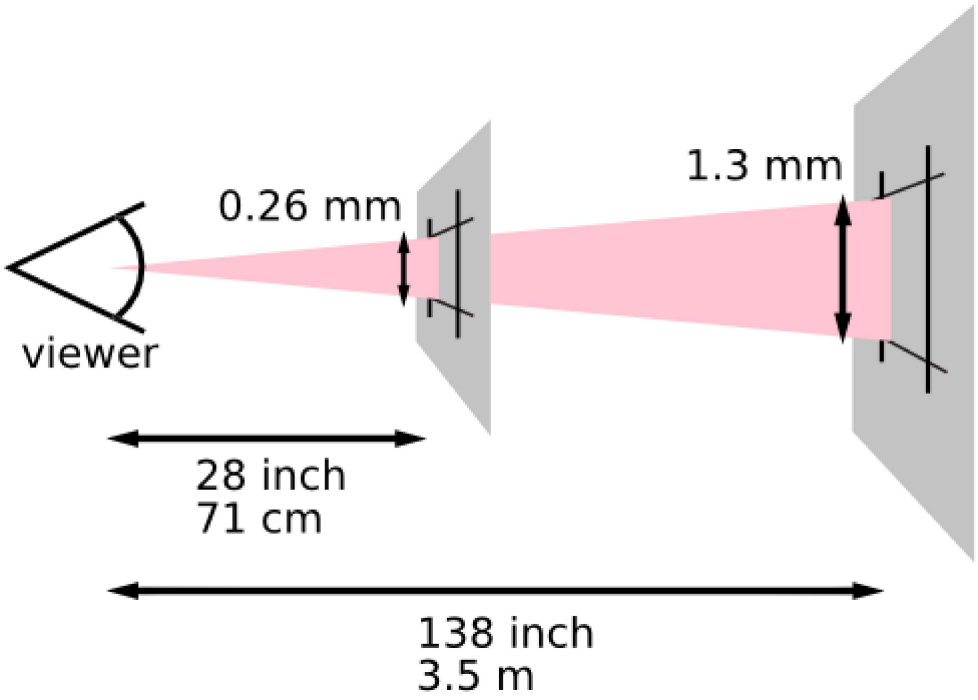


Figure 2 *Showing that pixels must become larger if the viewing distance increases*

This second image illustrates the effect of a device's resolution on the pixel unit: an area of 1px by 1px is covered by a single dot in a low-resolution device (e.g. a typical computer display), while the same area is covered by 16 dots in a higher resolution device (such as a printer).

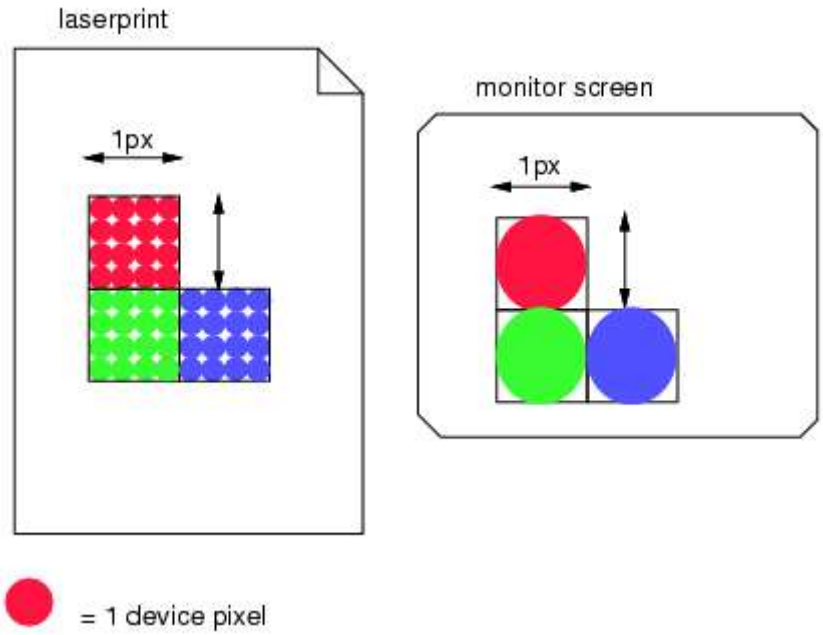


Figure 3 *Showing that more device pixels (dots) are needed to cover a 1px by 1px area on a high-resolution device than on a lower-resolution one (of the same approximate viewing distance)*

A ***device pixel*** is the smallest unit of area on the device output capable of displaying its full range of colors. For typical color screens, it's a square or somewhat rectangular region containing a red, green, and blue subpixel. Many non-traditional outputs exist that can blur this definition, such as by displaying some colors at higher resolutions. Such devices still expose some equivalent notion of "device pixel", however.

§ 7. Other Quantities

§ 7.1. Angle Units: the <angle> type and 'deg', 'grad', 'rad', 'turn' units

Angle values are <dimension>s denoted by ***'<angle>'***. The angle unit identifiers are:

'deg'
: Degrees. There are 360 degrees in a full circle.

'grad'
: Gradians, also known as "gons" or "grades". There are 400 gradians in a full circle.

'rad'
: Radians. There are 2π radians in a full circle.

'turn'
: Turns. There is 1 turn in a full circle.

For example, a right angle is '90deg' or '100grad' or '0.25turn' or approximately '1.57rad'.

All <angle> units are compatible, and 'deg' is their canonical unit.

> By convention, when an angle denotes a direction in CSS, it is typically interpreted as a ***bearing angle***, where 0deg is "up" or "north" on the screen, and larger angles are more clockwise (so 90deg is "right" or "east").
>
> For example, in the 'linear-gradient()' function, the <angle> that determines the direction of the gradient is interpreted as a bearing angle.

> Note: For legacy reasons, some uses of <angle> allow a bare '0' to mean '0deg'. This is not true in general, however, and will not occur in future uses of the <angle> type.

§ 7.2. Duration Units: the <time> type and 's', 'ms' units

Time values are dimensions denoted by ***‘<time>’***. The time unit identifiers are:

‘s’
: Seconds.

‘ms’
: Milliseconds. There are 1000 milliseconds in a second.

All <time> units are compatible, and ‘s’ is their canonical unit.

Properties may restrict the time value to some range. If the value is outside the allowed range, the declaration is invalid and must be ignored.

§ 7.3. Frequency Units: the <frequency> type and ‘Hz’, ‘kHz’ units

Frequency values are dimensions denoted by ***‘<frequency>’***. The frequency unit identifiers are:

‘Hz’
: Hertz. It represents the number of occurrences per second.

‘kHz’
: KiloHertz. A kiloHertz is 1000 Hertz.

For example, when representing sound pitches, 200Hz (or 200hz) is a bass sound, and 6kHz (or 6khz) is a treble sound.

All <frequency> units are compatible, and ‘hz’ is their canonical unit.

> Note: Units are ASCII case-insensitive and serialize as lower case, for example 1Hz serializes as 1hz.

§ 7.4. Resolution Units: the <resolution> type and ‘dpi’, ‘dpcm’, ‘dppx’ units

Resolution units are dimensions denoted by ***‘<resolution>’***. The resolution unit identifiers are:

‘dpi’
: Dots per inch.

‘dpcm’
: Dots per centimeter.

‘dppx’ , ***‘x’***

Dots per ‘px’ unit.

The <resolution> unit represents the size of a single "dot" in a graphical representation by indicating how many of these dots fit in a CSS ‘in’, ‘cm’, or ‘px’. For uses, see e.g. the ‘resolution’ media query in [MEDIAQ] or the ‘image-resolution’ property defined in [CSS3-IMAGES].

All <resolution> units are compatible, and ‘dppx’ is their canonical unit.

Note that due to the 1:96 fixed ratio of CSS ‘in’ to CSS ‘px’, ‘1dppx’ is equivalent to ‘96dpi’. This corresponds to the default resolution of images displayed in CSS: see ‘image-resolution’.

EXAMPLE 23
The following @media rule uses Media Queries [MEDIAQ] to assign some special style rules to devices that use two or more device pixels per CSS ‘px’ unit:

```
@media (min-resolution: 2dppx) { ... }
```

8. Data Types Defined Elsewhere

Some data types are defined in their own modules. This example talks about some of the most common ones used across several specifications.

8.1. Colors: the <color> type

The <color> data type is defined in [CSS-COLOR-4]. UAs must interpret <color> as defined therein.

8.1.1. Combination of <color>

Interpolation of <color> is defined in *CSS Color 4* § 13 Color Interpolation. Interpolation is done between premultiplied colors, as defined in *CSS Color 4* § 13.3 Interpolating with Alpha.

Addition of <color> is likewise defined as the independent addition of each component as a <number> in premultiplied space.

8.2. Images: the <image> type

The <image> data type is defined in [CSS3-IMAGES]. UAs that support CSS Images Level 3 or its successor must interpret <image> as defined therein. UAs that do not yet support CSS Images Level 3 must interpret <image> as <url>.

§ 8.2.1. Combination of <image>

Note: Interpolation of <image> is defined in *CSS Images 3* § 6 Interpolation.

Images are not additive.

§ 8.3. 2D Positioning: the <position> type

The **'<position>'** value specifies the position of a object area (e.g. background image) inside a positioning area (e.g. background positioning area). It is interpreted as specified for 'background-position'. [CSS3-BACKGROUND]

```
<position> = [
  [ left | center | right ] || [ top | center | bottom ]
|
  [ left | center | right | <length-percentage> ]
  [ top | center | bottom | <length-percentage> ]?
|
  [ [ left | right ] <length-percentage> ] &&
  [ [ top | bottom ] <length-percentage> ]
]
```

Note: The 'background-position' property also accepts a three-value syntax. This has been disallowed generically because it creates parsing ambiguities when combined with other length or percentage components in a property value.

§ 8.3.1. Parsing <position>

When specified in a grammar alongside other keywords, <length>s, or <percentage>s, <position> is *greedily* parsed; it consumes as many components as possible.

EXAMPLE 24

For example, ‘transform-origin’ defines a 3D position as (effectively) "<position> <length>?". A value such as ‘left 50px’ will be parsed as a 2-value <position>, with an omitted z-component; on the other hand, a value such as ‘top 50px’ will be parsed as a single-value <position> followed by a <length>.

§ 8.3.2. Serializing <position>

When serializing the specified value of a <position>:

↪ **If only one component is specified:**

- The implied ‘center’ keyword is added, and a 2-component value is serialized.

↪ **If two components are specified:**

- Keywords are serialized as keywords.
- <length-percentage>s are serialized as <length-percentage>s.
- Components are serialized horizontal first, then vertical.

↪ **If four components are specified:**

- Keywords and offsets are both serialized.
- Components are serialized horizontal first, then vertical.

Note: Computed values are always serialized as two offsets (without keywords) because the computed value does not preserve syntactic distinctions.

§ 8.3.3. Combination of <position>

Interpolation of <position> is defined as the independent interpolation of each component (x, y) normalized as an offset from the top left corner as a <length-percentage>.

Addition of <position> is likewise defined as the independent addition each component (x, y) normalized as an offset from the top left corner as a <length-percentage>.

§ 9. Functional Notations

A ***functional notation*** is a type of component value that can represent more complex types or invoke special processing. The syntax starts with the name of the function immediately followed by a left parenthesis (i.e. a <function-token>) followed by the argument(s) to the notation followed by a right parenthesis. White space is allowed, but optional, immediately inside the parentheses. Functions can take multiple arguments, which are formatted similarly to a CSS property value.

Some legacy functional notations, such as 'rgba()', use commas unnecessarily, but generally commas are only used to separate items in a list, or pieces of a grammar that would be ambiguous otherwise. If a comma is used to separate arguments, white space is optional before and after the comma.

> EXAMPLE 25
>
> ```
> background: url(http://www.example.org/image);
> color: rgb(100, 200, 50 );
> content: counter(list-item) ". ";
> width: calc(50% - 2em);
> ```

The math functions are defined in § 10 Mathematical Expressions.

§ 10. Mathematical Expressions

The ***math functions*** ('calc()', 'clamp()', 'sin()', and others defined in this chapter) allow numeric CSS values to be written as mathematical expressions.

A math function represents a numeric value, one of:

- <length>,
- <frequency>,
- <angle>,
- <time>,
- <flex>,
- <resolution>,
- <percentage>,
- <number>,
- <integer>

...or the <length-percentage>/etc mixed types, and can be used wherever such a value would be valid.

§ 10.1. Basic Arithmetic: 'calc()'

The ***'calc()'*** function is a math function that allows basic arithmetic to be performed on numerical values, using addition ('+'), subtraction ('-'), multiplication ('*'), division ('/'), and parentheses.

A 'calc()' function contains a single ***calculation***, which is a sequence of values interspersed with operators, and possibly grouped by parentheses (matching the <calc-sum> grammar), which represents the result of evaluating the expression using standard operator precedence rules ('*' and '/' bind tighter than '+' and '-', and operators are otherwise evaluated left-to-right). The 'calc()' function represents the result of its contained calculation.

Components of a calculation can be literal values (such as '5px'), other math functions, or other expressions, such as 'var()', that evaluate to a valid argument type (like <length>).

EXAMPLE 26

Math functions can be used to combine value that use different units. In this example the author wants the *margin box* of each section to take up 1/3 of the space, so they start with '100%/3', then subtract the element's borders and margins. ('box-sizing' can automatically achieve this effect for borders and padding, but a math function is needed if you want to include margins.)

```
section {
  float: left;
  margin: 1em; border: solid 1px;
  width: calc(100% / 3 - 2 * 1em - 2 * 1px);
}
```

Similarly, in this example the gradient will show a color transition only in the first and last '20px' of the element:

```
.fade {
  background-image: linear-gradient(silver 0%, white 20px,
                                    white calc(100% - 20px), silver 100%);
}
```

EXAMPLE 27

Math functions can also be useful just to express values in a more natural, readable fashion, rather than as an obscure decimal. For example, the following sets the 'font-size' so that exactly 35em fits within the viewport, ensuring that roughly the same amount of text always fills the screen no matter the screen size.

```
:root {
  font-size: calc(100vw / 35);
}
```

Functionality-wise, this is identical to just writing 'font-size: 2.857vw', but then the intent (that '35em' fills the viewport) is much less clear to someone reading the code; the later reader will have to reverse the math themselves to figure out that 2.857 is meant to approximate 100/35.

EXAMPLE 28

Standard mathematical precedence rules for the operators apply: 'calc(2 + 3 * 4)' is equal to '14', not '20'.

Parentheses can be used to manipulate precedence: 'calc((2 + 3) * 4)' is instead equal to '20'.

Parentheses and nesting additional 'calc()' functions are equivalent; the preceding expression could equivalently have been written as 'calc(calc(2 + 3) * 4)'. This can be useful when building up values piecemeal via 'var()', such as in the following example:

```
.aspect-ratio-box {
  --ar: calc(16 / 9);
  --w: calc(100% / 3);
  --h: calc(var(--w) / var(--ar));
  width: var(--w);
  height: var(--h);
}
```

Altho '--ar' *could* have been written as simply '--ar: (16 / 9);', '--w' is used both on its own (in 'width') and as a 'calc()' component (in '--h'), so it has to be written as a full 'calc()' function itself.

§ 10.2. Comparison Functions: ‘min()’, ‘max()’, and ‘clamp()’

The comparison functions of ‘min()’, ‘max()’, and ‘clamp()’ compare multiple calculations and represent the value of one of them.

The ***‘min()’*** or ***‘max()’*** functions contain one or more comma-separated calculations, and represent the smallest (most negative) or largest (most positive) of them, respectively.

The ***‘clamp()’*** function takes three calculations—a minimum value, a central value, and a maximum value—and represents its central calculation, clamped according to its min and max calculations, favoring the min calculation if it conflicts with the max. (That is, given ‘clamp(MIN, VAL, MAX)’, it represents exactly the same value as ‘max(MIN, min(VAL, MAX))’).

For all three functions, the argument calculations can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid; the result will have the same type as the arguments.

EXAMPLE 29

‘min()’, ‘max()’, and ‘clamp()’ can be used to make sure a value doesn’t exceed a "safe" limit: For example, "responsive type" that sets ‘font-size’ with viewport units might still want a minimum size to ensure readability:

```
.type {
  /* Set font-size to 10x the average of vw and vh,
     but don’t let it go below 12px. */
  font-size: max(10 * (1vw + 1vh) / 2, 12px);
}
```

Note: Full math expressions are allowed in each of the arguments; there’s no need to nest a ‘calc()’ inside! You can also provide more than two arguments, if you have multiple constraints to apply.

EXAMPLE 30

An occasional point of confusion when using ‘min()’/‘max()’ is that you use ‘max()’ to impose a minimum value on something (that is, properties like ‘min-width’ effectively use ‘max()’), and ‘min()’ to impose a maximum value on something; it’s easy to accidentally reach for the opposite function and try to use ‘min()’ to add a minimum size. Using ‘clamp()’ can make the code read more naturally, as the value is nestled between its minimum and maximum:

```
.type {
  /* Force the font-size to stay between 12px and 100px */
  font-size: clamp(12px, 10 * (1vw + 1vh) / 2, 100px);
}
```

Note that ‘clamp()’, matching CSS conventions elsewhere, has its minimum value "win" over its maximum value if the two are in the "wrong order". That is, ‘clamp(100px, ..., 50px)’ will resolve to ‘100px’, exceeding its stated "max" value.

If alternate resolution mechanics are desired they can be achieved by combining ‘clamp()’ with ‘min()’ or ‘max()’:

To have MAX win over MIN:

‘clamp(min(MIN, MAX), VAL, MAX)’. If you want to avoid repeating the MAX calculation, you can just reverse the nesting of functions that ‘clamp()’ is defined against—‘min(MAX, max(MIN, VAL))’.

To have MAX and MIN "swap" when they’re in the wrong order:

‘clamp(min(MIN, MAX), VAL, max(MIN, MAX))’. Unfortunately, there’s no easy way to do this without repeating the MIN and MAX terms.

§ 10.3. Stepped Value Functions: ‘round()’, ‘mod()’, and ‘rem()’

The stepped-value functions, ‘round()’, ‘mod()’, and ‘rem()’, all transform a given value according to another "step value", in different ways.

The ***‘round(<rounding-strategy>?, A, B)’*** function contains an optional rounding strategy, and two calculations A and B, and returns the value of A, rounded according to the rounding strategy, to the nearest integer multiple of B either above or below A. The argument calculations can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid; the result will have the same type as the arguments.

If A is exactly equal to an integer multiple of B, ‘round()’ resolves to A exactly (preserving whether A is 0^- or 0^+, if relevant). Otherwise, there are two integer multiples of B that are potentially "closest" to A, *lower B* which is closer to $-\infty$ and *upper B* which is closer to $+\infty$. The following ***‘<rounding-strategy>’***s dictate how to choose between them:

‘nearest’
: Choose whichever of *lower B* and *upper B* that has the smallest absolute difference from A. If both have an equal difference (A is exactly between the two values), choose *upper B*.

‘up’
: Choose *upper B*.

‘down’
: Choose *lower B*.

‘to-zero’
: Choose whichever of *lower B* and *upper B* that has the smallest absolute difference from 0.

If *lower B* would be zero, it is specifically equal to 0^+; if *upper B* would be zero, it is specifically equal to 0^-.

If <rounding-strategy> is omitted, it defaults to ‘nearest’. (Aka rounding to the nearest integer.)

ISSUE 5 CSSOM needs to specify how it rounds, and it’s probably good for CSS functions to round the same way by default. What behavior should be used? [Issue #5689]

EXAMPLE 31
Unlike languages like JavaScript which have a natural "precision" to round to (integers), CSS values have no such precision because values can be written in many different compatible units. As such, the precision has to be given explicitly; to round a width to the nearest ‘50px’, one can write ‘round(var(--width), 50px)’.

Note: JavaScript and other programming languages sometimes separate out the rounding strategies into separate rounding functions. JS’s `Math.floor()` is equivalent to CSS’s ‘round(down, ...)’; JS’s `Math.ceil()` is equivalent to CSS’s ‘round(up, ...)’; JS’s `Math.trunc()` is equivalent to CSS’s ‘round(to-zero, ...)’; and JS’s `Math.round()` is equivalent to CSS’s ‘round(nearest, ...)’, or just ‘round(...)’.

Note: The <rounding-strategy> keywords are the same as the keywords in 'block-step-size' and have the same behavior. ('block-step-size' just lacks 'to-zero'; since block sizes are always non-negative, 'to-zero' and 'down' would be identical.)

The modulus functions ***'mod(A, B)'*** and ***'rem(A, B)'*** similarly contain two calculations A and B, and return the difference between A and the nearest integer multiple of B either above or below A. The argument calculations can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid; the result will have the same type as the arguments.

The two functions are very similar, and in fact return identical results if both arguments are positive or both are negative: the value of the function is equal to the value of A shifted by the integer multiple of B that brings the value ***between zero and B***. (Specifically, the range includes zero and excludes B. More specifically, if B is positive the range starts at 0^+, and if B is negative it starts at 0^-.)

EXAMPLE 32

For example, 'mod(18px, 5px)' resolves to the value '3px', because subtracting '5px * 3' from '18px' yields '3px', which is the only such value between '0px' and '3px'.

Similarly, 'mod(-140deg, -90deg)' resolves to the value '-50deg', because adding '-90deg * 1' to '-140deg' yields '-50deg', which is the only such value between '0deg' and '-90deg'.

Evaluating either of these examples with 'rem()' yields the exact same results.

Their behavior diverges if the A value and the B step are on opposite sides of zero: 'mod()' (short for "modulus") continues to choose the integer multiple of B that puts the value between zero and B, as above (guaranteeing that the result will either be zero or share the sign of B, not A), while 'rem()' (short for "remainder") chooses the integer multiple of B that puts the value between zero and -B, avoiding changing the sign of the value.

EXAMPLE 33

For example, ‘mod(-18px, 5px)’ resolves to the value ‘2px’: adding ‘5px * 4’ to ‘-18px’ yields ‘2px’, which is between ‘0px’ and ‘5px’.

On the other hand, ‘rem(-18px, 5px)’ resolves to the value ‘-3px’: adding ‘5px * 3’ to ‘-18px’ yields ‘-3px’, which has the same sign as ‘-18px’ but is between ‘0px’ and ‘-5px’.

Similarly, ‘mod(140deg, -90deg)’ resolves to the value ‘-40deg’ (adding ‘-90deg * 2’ to ‘140deg’, bringing it to between ‘0deg’ and ‘-90deg’), but ‘rem(140deg, -90deg)’ resolves to the value ‘50deg’.

▶ **When should I choose ‘mod()’ vs ‘rem()’?**

Note: ‘mod()’ and ‘rem()’ can also be defined directly in terms of other functions: ‘mod(A, B)’ is equivalent to ‘calc(A - sign(B)*round(down, A*sign(B), B))’ (a hacky way to say "round(down) when B is positive, round(up) when B is negative), while ‘rem(A, B)’ is equivalent to ‘calc(A - round(to-zero, A, B))’. (These expressions don’t always handle 0^+ and 0^- correctly, though, because 0^- semantics aren’t commutative for addition.)

§ 10.3.1. Argument Ranges

In ‘round(A, B)’, if B is 0, the result is NaN. If A and B are both infinite, the result is NaN.

If A is infinite but B is finite, the result is the same infinity.

If A is finite but B is infinite, the result depends on the <rounding-strategy> and the sign of A:

‘nearest’ , **‘to-zero’**
: If A is positive or 0^+, return 0^+. Otherwise, return 0^-.

‘up’
: If A is positive (not zero), return $+\infty$. If A is 0^+, return 0^+. Otherwise, return 0^-.

‘down’
: If A is negative (not zero), return $-\infty$. If A is 0^-, return 0^-. Otherwise, return 0^+.

In ‘mod(A, B)’ or ‘rem(A, B)’, if B is 0, the result is NaN. If A is infinite, the result is NaN.

In ‘mod(A, B)’ only, if B is infinite and A has opposite sign to B (including an oppositely-signed zero), the result is NaN.

Note: All other "infinite B" cases are valid, and just return A immediately.

§ 10.4. Trigonometric Functions: 'sin()', 'cos()', 'tan()', 'asin()', 'acos()', 'atan()', and 'atan2()'

The trigonometric functions—'sin()', 'cos()', 'tan()', 'asin()', 'acos()', 'atan()', and 'atan2()'—compute the various basic trigonometric relationships.

The ***'sin(A)'***, ***'cos(A)'***, and ***'tan(A)'*** functions all contain a single calculation which must resolve to either a <number> or an <angle>, and compute their corresponding function by interpreting the result of their argument as radians. (That is, 'sin(45deg)', 'sin(.125turn)', and 'sin(3.14159 / 4)' all represent the same value, approximately '.707'.) They all represent a <number>; 'sin()' and 'cos()' will always return a number between −1 and 1, while 'tan()' can return any number between −∞ and +∞. (See § 10.9 Type Checking for details on how math functions handle ∞.)

The ***'asin(A)'***, ***'acos(A)'***, and ***'atan(A)'*** functions are the "arc" or "inverse" trigonometric functions, representing the inverse function to their corresponding "normal" trig functions. All of them contain a single calculation which must resolve to a <number>, and compute their corresponding function, interpreting their result as a number of radians, representing an <angle>. The angle returned by 'asin()' must be normalized to the range ['-90deg', '90deg']; the angle returned by 'acos()' to the range ['0deg', '180deg']; and the angle returned by 'atan()' to the range ['-90deg', '90deg'].

The ***'atan2(A, B)'*** function contains two comma-separated calculations, A and B. A and B can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid. The function returns the <angle> between the positive X-axis and the point (B,A). The returned angle must be normalized to the interval ('-180deg', '180deg'] (that is, greater than '-180deg', and less than or equal to '180deg').

Note: 'atan2(Y, X)' is *generally* equivalent to 'atan(Y / X)', but it gives a better answer when the point in question may include negative components. 'atan2(1, -1)', corresponding to the point (-1, 1), returns '135deg', distinct from 'atan2(-1, 1)', corresponding to the point (1, -1), which returns '-45deg'. In contrast, 'atan(1 / -1)' and 'atan(-1 / 1)' both return'-45deg', because the internal calculation resolves to '-1' for both.

§ 10.4.1. Argument Ranges

In ‘sin(A)’, ‘cos(A)’, or ‘tan(A)’, if A is infinite, the result is NaN. (See § 10.9 Type Checking for details on how math functions handle NaN.)

In ‘sin(A)’ or ‘tan(A)’, if A is 0^-, the result is 0^-.

In ‘tan(A)’, if A is one of the asymptote values (such as ‘90deg’, ‘270deg’, etc), the result must be $+\infty$ for ‘90deg’ and all values a multiple of ‘360deg’ from that (such as ‘-270deg’ or ‘450deg’), and $-\infty$ for ‘-90deg’ and all values a multiple of ‘360deg’ from that (such as ‘-450deg’ or ‘270deg’).

> Note: This is only relevant for units that can exactly represent the asymptotic values, such as ‘deg’ or ‘grad’. ‘rad’ cannot, and so whether the result is a very large negative or positive value can depend on rounding and precise details of how numbers are internally stored. It’s recommended you don’t depend on this behavior if using such units.

In ‘asin(A)’ or ‘acos(A)’, if A is less than -1 or greater than 1, the result is NaN.

In ‘acos(A)’, if A is exactly 1, the result is 0.

In ‘asin(A)’ or ‘atan(A)’, if A is 0^-, the result is 0^-.

In ‘atan(A)’, if A is $+\infty$, the result is ‘90deg’; if A is $-\infty$, the result is ‘-90deg’.

In ‘atan2(Y, X)’, the following table gives the results for all unusual argument combinations:

		X					
		$-\infty$	**-finite**	**0^-**	**0^+**	**+finite**	**$+\infty$**
Y	**$-\infty$**	-135deg	-90deg	-90deg	-90deg	-90deg	-45deg
	-finite	-180deg	(normal)	-90deg	-90deg	(normal)	0^-deg
	0^-	-180deg	-180deg	-180deg	0^-deg	0^-deg	0^-deg
	0^+	180deg	180deg	180deg	0^+deg	0^+deg	0^+deg
	+finite	180deg	(normal)	90deg	90deg	(normal)	0^+deg
	$+\infty$	135deg	90deg	90deg	90deg	90deg	45deg

Note: All of these behaviors are intended to match the "standard" definitions of these functions as implemented by most programming languages, in particular as implemented in JS.

Note: The behavior of 'tan(90deg)', while not constrained by JS behavior (because the JS function's input is in radians, and one cannot perfectly express a value of π/2 in JS numbers), is defined so that roundtripping of values works; 'tan(atan(infinity))' yields +∞, 'tan(atan(-infinity))' yields −∞, 'atan(tan(90deg))' yields '90deg', and 'atan(tan(-90deg))' yields '-90deg'.

§ 10.5. Exponential Functions: 'pow()', 'sqrt()', 'hypot()', 'log()', 'exp()'

The exponential functions—'pow()', 'sqrt()', 'hypot()', 'log()', and 'exp()'—compute various exponential functions with their arguments.

The ***'pow(A, B)'*** function contains two comma-separated calculations A and B, both of which must resolve to <number>s, and returns the result of raising A to the power of B, returning the value as a <number>.

The ***'sqrt(A)'*** function contains a single calculation which must resolve to a <number>, and returns the square root of the value as a <number>. ('sqrt(X)' and 'pow(X, .5)' are basically equivalent, differing only in some error-handling; 'sqrt()' is a common enough function that it is provided as a convenience.)

The ***'hypot(A, …)'*** function contains one or more comma-separated calculations, and returns the length of an N-dimensional vector with components equal to each of the calculations. (That is, the square root of the sum of the squares of its arguments.) The argument calculations can resolve to any <number>, <dimension>, or <percentage>, but must have the *same* type, or else the function is invalid; the result will have the same type as the arguments.

▶ **Why does 'hypot()' allow dimensions (values with units), but 'pow()' and 'sqrt()' only work on numbers?**

The ***'log(A, B?)'*** function contains one or two calculations (representing the value to be logarithmed, and the base of the logarithm, defaulting to e), which must resolve to <number>s, and returns the logarithm base B of the value A, as a <number>.

The ***'exp(A)'*** function contains one calculation which must resolve to a <number>, and returns the same value as 'pow(e, A)' as a <number>.

EXAMPLE 34

The ‘pow()’ function can be useful for strategies like CSS Modular Scale, which relates all the font-sizes on a page to each other by a fixed ratio.

These sizes can be easily written into custom properties like:

```
:root {
  --h6: calc(1rem * pow(1.5, -1));
  --h5: calc(1rem * pow(1.5, 0));
  --h4: calc(1rem * pow(1.5, 1));
  --h3: calc(1rem * pow(1.5, 2));
  --h2: calc(1rem * pow(1.5, 3));
  --h1: calc(1rem * pow(1.5, 4));
}
```

...rather than writing out the values in pre-calculated numbers like ‘5.0625rem’ (what ‘calc(1rem * pow(1.5, 4))’ resolves to) which have less clear provenance when encountered in a stylesheet.

EXAMPLE 35

With a single argument, ‘hypot()’ gives the absolute value of its input; ‘hypot(2em)’ and ‘hypot(-2em)’ both resolve to ‘2em’.

With more arguments, it gives the size of the main diagonal of a box whose side lengths are given by the arguments. This can be useful for transform-related things, giving the distance that an element will actually travel when it’s translated by a particular X, Y, and Z amount.

For example, ‘hypot(30px, 40px)’ resolves to ‘50px’, which is indeed the distance between an element’s starting and ending positions when it’s translated by a ‘translate(30px, 40px)’ transform. If an author wanted elements to get smaller as they moved further away from their starting point (drawing some sort of word cloud, for example), they could then use this distance in their scaling factor calculations.

EXAMPLE 36

With a single argument, 'log()' provides the "natural log" of its argument, or the log base e, same as JavaScript.

If one instead wants log base 10 (to, for example, count the number of digits in a value) or log base 2 (counting the number of bits in a value), 'log(X, 10)' or 'log(X, 2)' provide those values.

§ 10.5.1. Argument Ranges

In 'pow(A, B)', if A is negative and finite, and B is finite, B must be an integer, or else the result is NaN.

If A or B are infinite or 0, the following tables give the results:

	A is −∞	**A is 0⁻**	**A is 0⁺**	**A is +∞**
B is −finite	0⁻ if B is an odd integer, 0⁺ otherwise	−∞ if B is an odd integer, +∞ otherwise	+∞	0⁺
B is 0	always 1			
B is +finite	−∞ if B is an odd integer, +∞ otherwise	0⁻ if B is an odd integer, 0⁺ otherwise	0⁺	+∞

	A is < -1	**A is -1**	**-1 < A < 1**	**A is 1**	**A is > 1**
B is +∞	result is +∞	result is NaN	result is 0⁺	result is NaN	result is +∞
B is −∞	result is 0⁺	result is NaN	result is +∞	result is NaN	result is 0⁺

In 'sqrt(A)', if A is +∞, the result is +∞. If A is 0⁻, the result is 0⁻. If A is less than 0, the result is NaN.

In 'hypot(A, …)', if any of the inputs are infinite, the result is +∞.

In ‘log(A, B)’, if B is 1 or negative, B values *between* 0 and 1, or greater than 1, are valid. the result is NaN. If A is negative, the result is NaN. If A is 0^+ or 0^-, the result is $-\infty$. If A is 1, the result is 0^+. If A is $+\infty$, the result is $+\infty$.

In ‘exp(A)’, if A is $+\infty$, the result is $+\infty$. If A is $-\infty$, the result is 0^+.

(See § 10.9 Type Checking for details on how math functions handle NaN and infinities.)

> All of these behaviors are intended to match the "standard" definitions of these functions as implemented by most programming languages, in particular as implemented in JS.
> The only divergences from the behavior of the equivalent JS functions are that NaN is "infectious" in *every* function, forcing the function to return NaN if any argument calculation is NaN.
>
> ▶ **Details of the JS Behavior**

§ 10.6. Sign-Related Functions: ‘abs()’, ‘sign()’

The sign-related functions—‘abs()’ and ‘sign()’—compute various functions related to the sign of their argument.

The ***‘abs(A)’*** function contains one calculation A, and returns the absolute value of A, as the same type as the input: if A’s numeric value is positive or 0^+, just A again; otherwise ‘-1 * A’.

The ***‘sign(A)’*** function contains one calculation A, and returns -1 if A’s numeric value is negative, +1 if A’s numeric value is positive, 0^+ if A’s numeric value is 0^+, and 0^- if A’s numeric value is 0^-.

> Note: Both of these functions operate on the fully simplified/resolved form of their arguments, which may give unintuitive results at first glance. In particular, an expression like ‘10%’ might be positive *or* negative once it’s resolved, depending on what value it’s resolved against. For example, in ‘background-position’ positive percentages resolve to a negative length, and vice versa, if the background image is larger than the background area. Thus ‘sign(10%)’ might return ‘1’ *or* ‘-1’, depending on how the percentage is resolved! (Or even ‘0’, if it’s resolved against a zero length.)

§ 10.7. Numeric Constants: ‘e’, ‘pi’

While the trigonometric and exponential functions handle many complex numeric operations, some reasonable calculations must be put together more manually, and many times these include well-known constants, such as *e* and *π*.

Rather than require authors to manually type out several digits of these constants, a few of them are provided directly:

'e' is the base of the natural logarithm, approximately equal to 2.7182818284590452354.

'pi' is the ratio of a circle's circumference to its diameter, approximately equal to 3.1415926535897932.

> Note: These keywords are only usable within a calculation, such as 'calc(pow(e, pi) - pi)', or 'min(pi, 5, e)'. If used outside of a calculation, they're treated like any other keyword: 'animation-name: pi;' refers to an animation named "pi"; 'line-height: e;' is invalid (*not* similar to 'line-height: 2.7', but 'line-height: calc(e);' is).

§ 10.7.1. Degenerate Numeric Constants: 'infinity', '-infinity', 'NaN'

When a calculation or a subtree of a calculation becomes infinite or NaN, representing it with a numeric value is no longer possible. To aid in serialization of these degenerate values, the additional math constants ***'infinity'*** (with the value +∞), ***'-infinity'*** (with the value −∞), and ***'NaN'*** (with the value NaN) are defined.

As usual for CSS keywords, these are ASCII case-insensitive. Thus, 'calc(InFiNiTy)' is perfectly valid. However, 'NaN' must be serialized with this canonical casing.

> Note: While not *technically* numbers, these keywords act as numeric values, similar to 'e' and 'pi'. Thus to get an infinite length, for example, requires an expression like 'calc(infinity * 1px)'.

> Note: These constants are defined *mostly* to make serialization of infinite/NaN values simpler and more obvious, but *can* be used to indicate a "largest possible value", since an infinite value gets clamped to the allowed range. It's rare for this to be reasonable, but when it is, using 'infinity' is clearer in its intent than just putting an enormous number in one's stylesheet.

§ 10.8. Syntax

The syntax of a math function is:

```
<calc()>  = calc( <calc-sum> )
<min()>   = min( <calc-sum># )
<max()>   = max( <calc-sum># )
<clamp()> = clamp( <calc-sum>#{3} )
<round()> = round( <rounding-strategy>?, <calc-sum>, <calc-sum> )
<mod()>   = mod( <calc-sum>, <calc-sum> )
<rem()>   = rem( <calc-sum>, <calc-sum> )
<sin()>   = sin( <calc-sum> )
<cos()>   = cos( <calc-sum> )
<tan()>   = tan( <calc-sum> )
<asin()>  = asin( <calc-sum> )
<acos()>  = acos( <calc-sum> )
<atan()>  = atan( <calc-sum> )
<atan2()> = atan2( <calc-sum>, <calc-sum> )
<pow()>   = pow( <calc-sum>, <calc-sum> )
<sqrt()>  = sqrt( <calc-sum> )
<hypot()> = hypot( <calc-sum># )
<log()>   = log( <calc-sum>, <calc-sum>? )
<exp()>   = exp( <calc-sum> )
<abs()>   = abs( <calc-sum> )
<sign()>  = sign( <calc-sum> )
<calc-sum> = <calc-product> [ [ '+' | '-' ] <calc-product> ]*
<calc-product> = <calc-value> [ [ '*' | '/' ] <calc-value> ]*
<calc-value> = <number> | <dimension> | <percentage> |
               <calc-constant> | ( <calc-sum> )
<calc-constant> = e | pi | infinity | -infinity | NaN
<rounding-strategy> = nearest | up | down | to-zero
```

In addition, whitespace is required on both sides of the '+' and '-' operators. (The '*' and '/' operators can be used without white space around them.)

Several of the math functions above have additional constraints on what their <calc-sum> arguments can contain. Check the definitions of the individual functions for details.

UAs must support calculations of at least 32 <calc-value> terms and at least 32 levels of nesting (parentheses and/or functions). For functions that support an arbitrary number of arguments (such as 'min()'), it must also support at least 32 arguments. If a calculation contains more than the supported number of terms, arguments, or nesting it must be treated as if it were invalid.

§ 10.9. Type Checking

A math function can be many possible types, such as <length>, <number>, etc., depending on the calculations it contains, as defined below. It can be used anywhere a value of that type is allowed.

> EXAMPLE 37
> For example, the 'width' property accepts <length> values, so a math function that resolves to a <length>, such as 'calc(5px + 1em)', can be used in 'width'.

Additionally, math functions that resolve to <number> can be used in any place that only accepts <integer>; the value is rounded to the nearest integer as it resolves.

Operators form sub-expressions, which gain types based on their arguments.

> Note: In previous versions of this specification, multiplication and division were limited in what arguments they could take, to avoid producing more complex intermediate results (such as '1px * 1em', which is <length>²) and to make division-by-zero detectable at parse time. This version now relaxes those restrictions.

To ***determine the type of a calculation***:

- At a '+' or '-' sub-expression, attempt to add the types of the left and right arguments. If this returns failure, the entire calculation's type is failure. Otherwise, the sub-expression's type is the returned type.
- At a '*' sub-expression, multiply the types of the left and right arguments. The sub-expression's type is the returned result.
- At a '/' sub-expression, let *left type* be the result of finding the types of its left argument, and *right type* be the result of finding the types of its right argument and then inverting it.

  The sub-expression's type is the result of multiplying the *left type* and *right type*.
- Anything else is a terminal value, whose type is determined based on its CSS type:

  ↪ **<number>**
  ↪ **<integer>**
  the type is «[ ]» (empty map)

  ↪ **<length>**
  the type is «["length" → 1]»

  ↪ **<angle>**
  the type is «["angle" → 1]»

  ↪ **<time>**

the type is «["time" → 1]»

↪ **<frequency>**
the type is «["frequency" → 1]»

↪ **<resolution>**
the type is «["resolution" → 1]»

↪ **<flex>**
the type is «["flex" → 1]»

↪ **<calc-constant>**
the type is «[ ]» (empty map)

↪ **<percentage>**
If, in the context in which the math function containing this calculation is placed, <percentage>s are resolved relative to another type of value (such as in 'width', where <percentage> is resolved against a <length>), and that other type is *not* <number>, the type is determined as the other type.

Otherwise, the type is «["percent" → 1]».

↪ **anything else**
The calculation's type is failure.

In all cases, the associated percent hint is null.

Math functions themselves have types, according to their contained calculations:

'calc()', **'abs()'**
The type of its contained calculation.

'min()', **'max()'**, **'clamp()'**, **'hypot()'**, **'round()'**, **'mod()'**, **'rem()'**
The result of adding the types of its comma-separated calculations.

'sign()', **'sin()'**, **'cos()'**, **'tan()'**, **'asin()'**, **'acos()'**, **'atan()'**, **'atan2()'**, **'pow()'**, **'sqrt()'**, **'log()'**, **'exp()'**
«[ ]» (empty map).

For each of the above, if the type is failure, the math function is invalid.

A math function resolves to <number>, <length>, <angle>, <time>, <frequency>, <resolution>, <flex>, or <percentage> according to which of those productions its type matches. (These categories are mutually exclusive.) If it can't match any of these, the math function is invalid.

Division by zero is possible, which introduces certain complications. Math functions follow IEEE-754 semantics for these operations:

- Dividing a positive value by zero produces +∞.
- Dividing a negative value by zero produces −∞.
- Adding or subtracting ±∞ to anything produces the appropriate infinity, unless a following rule would define it as producing NaN.
- Multiplying any value by ±∞ produces the appropriate infinity, unless a following rule would define it as producing NaN.
- Dividing any value by ±∞ produces zero, unless a following rule would define it as producing NaN.
- Dividing zero by zero, dividing ±∞ by ±∞, multiplying 0 by ±∞, adding +∞ to −∞ (or the equivalent subtractions) produces NaN.
- Any operation with at least one NaN argument produces NaN.

Additionally, IEEE-754 introduces the concept of "negative zero", which must be tracked within a calculation and between nested calculations:

- Negative zero (0^-) can be produced by a multiplication or division that produces zero with exactly one negative argument (such as '-5 * 0' or '1 / (-infinity)'), or by certain argument combinations in the other math functions.

  > Note: Note that negative zeros don't escape a math function; as detailed below, they're "censored" away into an "unsigned" zero.

- '$0^- + 0^-$' or '$0^- - 0$' produces 0^-. All other additions or subtractions that would produce a zero produce 0^+.
- Multiplying or dividing 0^- with a positive number (including 0^+) produces a negative result (either 0^- or −∞), while multiplying or dividing 0^- with a negative number produces a positive result.

  (In other words, multiplying or dividing with 0^- follows standard sign rules.)

- When comparing 0^+ and 0^-, 0^- is less than 0^+. For example, '$\min(0^+, 0^-)$' must produce 0^-, '$\max(0^+, 0^-)$' must produce 0^+, and '$\text{clamp}(0^+, 0^-, 1)$' must produce 0^+.

If a ***top-level calculation*** (a math function not nested inside of another math function) would produce a value whose numeric part is NaN, it instead act as though the numeric part is 0. If a top-level

calculation would produce a value whose numeric part is 0^-, it instead acts as though the numeric part is the standard "unsigned" zero.

EXAMPLE 38

For example, 'calc(-5 * 0)' produces an unsigned zero—the calculation resolves to 0^-, but as it's a top-level calculation, it's then censored to an unsigned zero.

On the other hand, 'calc(1 / calc(-5 * 0))' produces $-\infty$, same as 'calc(1 / (-5 * 0))'—the inner calc resolves to 0^-, and as it's not a top-level calculation, it passes it up unchanged to the outer calc to produce $-\infty$. If it was censored into an unsigned zero, it would instead produce $+\infty$.

Note: Algebraic simplifications do not affect the validity of a math function or its resolved type. For example, 'calc(5px - 5px + 10s)' and 'calc(0 * 5px + 10s)' are both invalid due to the attempt to add a length and a time.

Note: Note that <percentage>s relative to <number>s, such as in 'opacity', are not *combinable* with those numbers—'opacity: calc(.25 + 25%)' is invalid. Allowing this causes significant problems with "unit algebra" (allowing multiplication/division of <dimension>s), and in every case so far, doesn't provide any new functionality. (For example, 'opacity: 25%' is identical to 'opacity: .25'; it's just a trivial syntax transform.) You can still perform other operations with them, such as 'opacity: calc(100% / 3);', which is valid.

Note: Because <number-token>s are always interpreted as <number>s or <integer>s, "unitless 0" <length>s aren't supported in math functions. That is, 'width: calc(0 + 5px);' is invalid, because it's trying to add a <number> to a <length>, even though both 'width: 0;' and 'width: 5px;' are valid.

Note: Altho there are a few properties in which a bare <number> becomes a <length> at used-value time (specifically, 'line-height' and 'tab-size'), <number>s never become "length-like" in 'calc()'. They always stay as <number>s.

Note: In Quirks Mode [QUIRKS], some properties that would normally only accept <length>s are defined to also accept <number>s, interpreting them as 'px' lengths. Like unitless zeroes, this has no effect on the parsing or behavior of math functions, tho a math function that resolves to a <number> value might become valid in Quirks Mode (and have its result interpreted as a 'px' length).

§ 10.10. Internal Representation

The internal representation of a math function is a ***calculation tree***: a tree where the branch nodes are ***operator nodes*** corresponding either to math functions (such as Min, Cos, Sqrt, etc) or to operators in a calculation (Sum, Product, Negate, and Invert, the ***calc-operator nodes***), and the leaf nodes are either numeric values (such as numbers, dimensions, and percentages) or non-math functions that resolve to a numeric type.

Math functions are turned into calculation trees depending on the function:

↪ **calc()**
: The internal representation of a 'calc()' function is the result of parsing a calculation from its argument.

↪ **any other math function**
: The internal representation is an operator node with the same name as the function, whose children are the result of parsing a calculation from each of the function's arguments, in the order they appear.

To ***parse a calculation***, given a calculation *values* represented as a list of component values, and returning a calculation tree:

1. Discard any <whitespace-token>s from *values*.
2. An item in *values* is an "operator" if it's a <delim-token> with the value "+", "-", "*", or "/". Otherwise, it's a "value".
3. Collect children into Product and Invert nodes.

   For every consecutive run of value items in *values* separated by "*" or "/" operators:

   > 1. For each "/" operator in the run, replace its right-hand value item *rhs* with an Invert node containing *rhs* as its child.
   > 2. Replace the entire run with a Product node containing the value items of the run as its children.

4. Collect children into Sum and Negate nodes.

   > 1. For each "-" operator item in *values*, replace its right-hand value item *rhs* with a Negate node containing *rhs* as its child.
   > 2. If *values* has only one item, and it is a Product node or a parenthesized simple block, replace *values* with that item.

Otherwise, replace *values* with a Sum node containing the value items of *values* as its children.

5. At this point *values* is a tree of Sum, Product, Negate, and Invert nodes, with other types of values at the leaf nodes. Process the leaf nodes.

   For every leaf node *leaf* in *values*:

   1. If *leaf* is a parenthesized simple block, replace *leaf* with the result of parsing a calculation from *leaf*'s contents.
   2. If *leaf* is a math function, replace *leaf* with the internal representation of that math function.

6. Return the result of simplifying a calculation tree from *values*.

§ 10.10.1. Simplification

Internal representations of math functions are eagerly simplified to the extent possible, using standard algebraic simplifications (distributing multiplication over sums, combining similar units, etc.).

To ***simplify a calculation tree*** *root*:

1. If *root* is a numeric value:

   1. If *root* is a percentage that will be resolved against another value, and there is enough information available to resolve it, do so, and express the resulting numeric value in the appropriate canonical unit. Return the value.
   2. If *root* is a dimension that is not expressed in its canonical unit, and there is enough information available to convert it to the canonical unit, do so, and return the value.
   3. If *root* is a <calc-constant>, return its numeric value.
   4. Otherwise, return *root*.

2. If *root* is any other leaf node (not an operator node):

   1. If there is enough information available to determine its numeric value, return its value, expressed in the value's canonical unit.
   2. Otherwise, return *root*.

3. At this point, *root* is an operator node. Simplify all the calculation children of *root*.
4. If *root* is an operator node that's not one of the calc-operator nodes, and all of its calculation children are numeric values with enough information to compute the operation *root* represents,

return the result of running *root*'s operation using its children, expressed in the result's canonical unit.

> If a percentage is left at this point, it will *usually* block simplification of the node, since it needs to be resolved against another value using information not currently available. (Otherwise, it would have been converted to a different value in an earlier step.) This includes operations such as "min", since percentages might resolve against a negative basis, and thus end up with an opposite comparative relationship than the raw percentage value would seem to indicate.
>
> However, "raw" percentages—ones which do not resolve against another value, such as in 'opacity'—might not block simplification.

5. If *root* is a Min or Max node, attempt to *partially* simplify it:

    1. For each node *child* of *root*'s children:

        If *child* is a numeric value with enough information to compare magnitudes with another child of the same unit (see note in previous step), and there are other children of *root* that are numeric values with the same unit, combine all such children with the appropriate operator per *root*, and replace *child* with the result, removing all other child nodes involved.

    2. Return *root*.

6. If *root* is a Negate node:

    1. If *root*'s child is a numeric value, return an equivalent numeric value, but with the value negated (0 - value).
    2. If *root*'s child is a Negate node, return the child's child.
    3. Return *root*.

7. If *root* is an Invert node:

    1. If *root*'s child is a number (not a percentage or dimension) return the reciprocal of the child's value.
    2. If *root*'s child is an Invert node, return the child's child.
    3. Return *root*.

8. If *root* is a Sum node:

    1. For each of *root*'s children that are Sum nodes, replace them with their children.

2. For each set of *root*'s children that are numeric values with identical units, remove those children and replace them with a single numeric value containing the sum of the removed nodes, and with the same unit.

   (E.g. combine numbers, combine percentages, combine px values, etc.)

3. If *root* has only a single child at this point, return the child. Otherwise, return *root*.

9. If *root* is a Product node:

   1. For each of *root*'s children that are Product nodes, replace them with their children.
   2. If *root* has multiple children that are numbers (not percentages or dimensions), remove them and replace them with a single number containing the product of the removed nodes.
   3. If *root* contains only two children, one of which is a number (not a percentage or dimension) and the other of which is a Sum whose children are all numeric values, multiply all of the Sum's children by the number, then return the Sum.
   4. If *root* contains only numeric values and/or Invert nodes containing numeric values, and multiplying the types of all the children (noting that the type of an Invert node is the inverse of its child's type) results in a type that matches any of the types that a math function can resolve to, return the result of multiplying all the values of the children (noting that the value of an Invert node is the reciprocal of its child's value), expressed in the result's canonical unit.
   5. Return *root*.

§ 10.11. Computed Value

The computed value of a math function is its calculation tree simplified, using all the information available at computed value time. (Such as the 'em' to 'px' ratio, how to resolve percentages in some properties, etc.)

Where percentages are not resolved at computed-value time, they are not resolved in math functions, e.g. 'calc(100% - 100% + 1px)' resolves to 'calc(0% + 1px)', not to '1px'. If there are special rules for computing percentages in a value (e.g. the 'height' property), they apply whenever a math function contains percentages.

The calculation tree is again simplified at used value time; with used value time information, a math function always simplifies down to a single numeric value.

EXAMPLE 39

For example, whereas ‘font-size’ computes percentage values at computed value time so that font-relative length units can be computed, ‘background-position’ has layout-dependent behavior for percentage values, and thus does not resolve percentages until used-value time.

Due to this, ‘background-position’ computation preserves the percentage in a ‘calc()’ whereas ‘font-size’ will compute such expressions directly into a length.

Given the complexities of width and height calculations on table cells and table elements, math expressions mixing both percentages and non-zero lengths for widths and heights on table columns, table column groups, table rows, table row groups, and table cells in both auto and fixed layout tables MUST be treated as if ‘auto’ had been specified.

§ 10.12. Range Checking

Parse-time range-checking of values is not performed within math functions, and therefore out-of-range values do not cause the declaration to become invalid. However, the value resulting from an expression must be clamped to the range allowed in the target context. Clamping is performed on computed values to the extent possible, and also on used values if computation was unable to sufficiently simplify the expression to allow range-checking. (Clamping is not performed on specified values.)

Note: This requires all contexts accepting ‘calc()’ to define their allowable values as a closed (not open) interval.

Note: By definition, ±∞ are outside the allowed range for any property, and will clamp to the minimum/maximum value allowed. Even properties that can explicitly represent infinity as a keyword value, such as ‘animation-iteration-count’, will end up clamping ±∞, as math functions can’t resolve to keyword values; the *numeric* part of the property’s syntax still has a minimum/maximum value.

Additionally, if a math function that resolves to <number> is used somewhere that only accepts <integer>, the computed value and used value are rounded to the nearest integer, in the same manner as clamping, above.

EXAMPLE 40

Since widths smaller than 0px are not allowed, these three declarations are equivalent:

```
width: calc(5px - 10px);
width: calc(-5px);
width: 0px;
```

Note however that 'width: -5px' is not equivalent to 'width: calc(-5px)'! Out-of-range values *outside* 'calc()' are syntactically invalid, and cause the entire declaration to be dropped.

§ 10.13. Serialization

ISSUE 6 This section is still under discussion.

To ***serialize a math function*** *fn*:

1. If the root of the calculation tree *fn* represents is a numeric value (number, percentage, or dimension), and the serialization being produced is of a computed value or later, then clamp the value to the range allowed for its context (if necessary), then serialize the value as normal and return the result.
2. If *fn* represents an infinite or NaN value:

    1. Let *s* be the string "calc(".
    2. Serialize the keyword 'infinity', '-infinity', or 'NaN', as appropriate to represent the value, and append it to *s*.
    3. If *fn*'s type is anything other than «[ ]» (empty, representing a <number>), append " * " to *s*. Create a numeric value in the canonical unit for *fn*'s type (such as 'px' for <length>), with a value of 1. Serialize this numeric value and append it to *s*.
    4. Return *s*.
3. If the calculation tree's root node is a numeric value, or a calc-operator node, let *s* be a string initially containing "calc(".

    Otherwise, let *s* be a string initially containing the name of the root node, lowercased (such as "sin" or "max"), followed by a "(" (open parenthesis).
4. For each child of the root node, serialize the calculation tree. If a result of this serialization starts with a "(" (open parenthesis) and ends with a ")" (close parenthesis), remove those characters

from the result. Concatenate all of the results using ", " (comma followed by space), then append the result to *s*.

5. Append ")" (close parenthesis) to *s*.
6. Return *s*.

To ***serialize a calculation tree***:

1. Let *root* be the root node of the calculation tree.
2. If *root* is a numeric value, or a non-math function, serialize *root* per the normal rules for it and return the result.
3. If *root* is anything but a Sum, Negate, Product, or Invert node, serialize a math function for the function corresponding to the node type, treating the node's children as the function's comma-separated calculation arguments, and return the result.
4. If *root* is a Negate node, let *s* be a string initially containing "(-1 * ".

   Serialize *root*'s child, and append it to *s*.

   Append ")" to *s*, then return it.
5. If *root* is an Invert node, let *s* be a string initially containing "(1 / ".

   Serialize *root*'s child, and append it to *s*.

   Append ")" to *s*, then return it.
6. If *root* is a Sum node, let *s* be a string initially containing "(".

   Sort root's children.

   Serialize *root*'s first child, and append it to *s*.

   For each *child* of *root* beyond the first:

   > 1. If *child* is a Negate node, append " - " to *s*, then serialize the Negate's child and append the result to *s*.
   > 2. If *child* is a negative numeric value, append " - " to *s*, then serialize the negation of *child* as normal and append the result to *s*.
   > 3. Otherwise, append " + " to *s*, then serialize *child* and append the result to *s*.

   Finally, append ")" to *s* and return it.
7. If *root* is a Product node, let *s* be a string initially containing "(".

Sort root's children.

Serialize *root*'s first child, and append it to *s*.

For each *child* of *root* beyond the first:

1. If *child* is an Invert node, append " / " to *s*, then serialize the Invert's child and append the result to *s*.
2. Otherwise, append " * " to *s*, then serialize *child* and append the result to *s*.

Finally, append ")" to *s* and return it.

To ***sort a calculation's children*** *nodes*:

1. Let *ret* be an empty list.
2. If *nodes* contains a number, remove it from *nodes* and append it to *ret*.
3. If *nodes* contains a percentage, remove it from *nodes* and append it to *ret*.
4. If *nodes* contains any dimensions, remove them from *nodes*, sort them by their units, ordered ASCII case-insensitively, and append them to *ret*.
5. If *nodes* still contains any items, append them to *ret* in the same order.
6. Return *ret*.

EXAMPLE 41

For example, 'calc(20px + 30px)' would serialize as 'calc(50px)' as a specified value, or as '50px' as a computed value.

A value like 'calc(20px + 0%)' would serialize as 'calc(0% + 20px)', maintaining both terms in the serialized value. (It's important to maintain zero-valued terms, so the 'calc()' doesn't suddenly "change shape" in the middle of a transition when one of the values happens to have a zero value temporarily. This also removes the need to "pick a unit" when all the terms are zero.)

A value like 'calc(20px + 2em)' would serialize as 'calc(2em + 20px)' as a specified value (maintaining both units as they're incompatible at specified-value time, but sorting them alphabetically), or as something like '52px' as a computed value ('em' values are converted to absolute lengths at computed-value time, so assuming '1em' = '16px', they combine into '52px', which then drops the 'calc()' wrapper.)

See [CSSOM] for further information on serialization.

§ 10.14. Combination of Math Functions

Interpolation of math functions, with each other or with numeric values and other numeric-valued functions, is defined as V_{result} = calc((1 - p) * V_a + p * V_b). (Simplification of the value might then reduce the expression to a smaller, simpler form.)

Addition of math functions, with each other or with numeric values and other numeric-valued functions, is defined as V_{result} = calc(V_a + V_b). (Simplification of the value might then reduce the expression to a smaller, simpler form.)

§ Appendix A: Coordinating List-Valued Properties

Some list-valued properties have coordinated effects: each item in their value list applies to a distinct effect, and corresponding entries in each property's list all refer to the same effect. Often the coordinating values can also be specified together as a single entry in a list-valued shorthand property.

A typical example is the list-valued 'background-*' properties, which can specify multiple background image layers. For each property controlling how the image is sized, tiled, placed, etc., the *N*th item in its list describes some effect that applies to the *N*th background image.

A ***coordinating list property group*** creates a ***coordinated value list***, which has, for each entry, a value from each property in the group; these are used together to define a single effect, such as a background image layer or an animation. The coordinated value list is assembled as follows:

- The length of the coordinated value list is determined by the number of items specified in one particular coordinating list property, the ***coordinating list base property***. (In the case of backgrounds, this is the 'background-image' property.)
- The *N*th value of the coordinated value list is constructed by collecting the *N*th use value of each coordinating list property
- If a coordinating list property has too many values specified, excess values at the end of its list are not used.
- If a coordinating list property has too few values specified, its value list is repeated to add more used values.
- The computed values of the coordinating list properties are not affected by such truncation or repetition.

§ Appendix B: IANA Considerations

§ Registration for the `about:invalid` URL scheme

This sections defines and registers the `about:invalid` URL, in accordance with the registration procedure defined in [RFC6694].

The official record of this registration can be found at http://www.iana.org/assignments/about-uri-tokens/about-uri-tokens.xhtml.

Registered Token	`invalid`
Intended Usage	The `about:invalid` URL references a non-existent document with a generic error condition. It can be used when a URL is necessary, but the default value shouldn't be resolvable as any type of document.
Contact/Change controller	CSS WG <www-style@w3.org> (on behalf of W3C)
Specification	CSS Values and Units Module Level 3

§ Appendix C: Quirky Lengths

When CSS is being parsed in quirks mode, ***'<quirky-length>'*** is a type of <length> that is only valid in certain properties:

- 'background-position'
- 'border-spacing'
- 'border-top-width'
- 'border-right-width'
- 'border-bottom-width'
- 'border-left-width'
- 'border-width'
- 'bottom'
- 'clip'
- 'font-size'

- ‘height’
- ‘left’
- ‘letter-spacing’
- ‘margin-right’
- ‘margin-left’
- ‘margin-top’
- ‘margin-bottom’
- ‘margin’
- ‘max-height’
- ‘max-width’
- ‘min-height’
- ‘min-width’
- ‘padding-top’
- ‘padding-right’
- ‘padding-bottom’
- ‘padding-left’
- ‘padding’
- ‘right’
- ‘text-indent’
- ‘top’
- ‘vertical-align’
- ‘width’
- ‘word-spacing’

It is *not* valid in properties that include or reference these properties, such as the ‘background’ shorthand, or inside functional notations such as ‘calc()’, except that they must be allowed in ‘rect()’ in the ‘clip’ property.

Additionally, while <quirky-length> must be valid as a <length> when parsing the affected properties in the ‘@supports’ rule, it is *not* valid for those properties when used in the `CSS.supports()` method.

A <quirky-length> is syntactically identical to a <number-token>, and is interpreted as a 'px' length with the same value.

(In other words, Quirks Mode allows all 'px' lengths in the affected properties to be written without a unit, similar to unitless zero lengths.)

§ Acknowledgments

Firstly, the editors would like to thank all of the contributors to the previous level of this module.

Secondly, we would like to acknowledge Anthony Frehner, Emilio Cobos Álvarez, Koji Ishii, Noam Rosenthal, and Xidorn Quan for their comments and suggestions, which have improved Level 4.

§ Changes

§ Recent Changes

(This is a subset of Additions Since Level 3.)

Substantial changes since 16 December 2021 WD:

- Changed resolution of a 'url()' with the local url flag to reference the current node tree (regardless of document base URL modifications). (Issue 3320)
- Switched censoring of 'NaN' that escapes a math function from infinity to zero. (Issue 7067)
- Added Appendix A: Coordinating List-Valued Properties to allow this property pattern to be easily referenced. (Issue 7164)
- Restricted 'mix()' to be the sole value of a declaration. (Issue 6700)
- Updated to match latest Fetch terminology. (Fetch PR 1413, CSS PR 7160)
- Clarified that the font-relative lengths are calculated without text shaping.
- Defined serialization of empty urls to be `url("")`. (Issue 6447)
- Defined serialization of <position> specified values. (Issue 2274)
- Fixed definition of numbers to allow decimals in combination with scientific notation, as originally intended and as defined in [CSS-SYNTAX-3]. (Issue 7248)
- Corrected various functions to return an empty map for their type instead of «["number" → 1]». (Issue 7486)

- Clarified effect of special UA restrictions on ‘line-height’ on ‘lh’ and ‘rlh’. (Issue 3257)
- Defined `<function()>` notation to refer to functional notations. (Issue 5728)

Substantial changes since 16 October 2021 WD:

- Switched ‘*vi’ and ‘*vb’ units to resolve against the computed writing mode of the element itself. (Issue 6873)
- Added § 4.5.4 URL Processing Model to define integration with CORS, etc. (Issue 562)
- Fixed the inverted assignment of viewport-percentage length behaviors to types of interface changes (A vs. B).

  > - Changes in interface that happen as a result of scrolling or other frequent page interactions that would disturb the user if they resulted in substantial layout changes must be categorized as the ~~former (A)~~ latter (B) .
  > - Changes in interface that have a sufficiently steady state that re-laying out the document into the adjusted space would be beneficial to the user must be categorized as the ~~latter (B)~~ former (A) .

- Defined minimum number of ‘calc()’ terms, arguments, and nesting as 32. (Issue 3462)
- Defined that ‘mod(-0, infinity)’ returns ‘NaN’. (Issue 4723)
- Deferred ‘toggle()’ and ‘attr()’ to Level 5.

Changes since 30 September 2021 WD:

- Added ‘rex’, ‘rcap’, ‘rch’, and ‘ric’ units.
- Switched ‘toggle()’ to use semicolons, matching with ‘mix()’. (Issue 6701)
- Fixed some wording errors in the definition of ‘calc()’. (Issue 6506)
- Imported definition of <quirky-length> from [QUIRKS]. (Issue 6100)

Changes since 7 July 2021 WD:

- Added ‘mix()’ notation for representing interpolated values.
- Defined generically the computation of <integer>, <number>, <percentage>, and <length>.
- Clarified that only non-zero lengths create a percentage+length mix that switches table cells to ‘auto’ sizing.

Changes since 11 November 2020 WD:

- Updated interpolation of colors to reference [CSS-COLOR-4] instead of [CSS-COLOR-3].
- Added the ‘svh’, ‘svw’, ‘svi’, ‘svb’, ‘svmin’, and ‘svmax’ small viewport-percentage units; ‘lvh’, ‘lvw’, ‘lvi’, ‘lvb’, ‘lvmin’, and ‘lvmax’ large viewport-percentage units; and ‘dvh’, ‘dvw’, ‘dvi’, ‘dvb’, ‘dvmin’, and ‘dvmax’ dynamic viewport-percentage units. (Issue 4329 and Issue 6113)
- Clamped excessively large <angle> values to multiples of ‘360deg’. (Issue 6105)
- Added back rules on range-checking combined values lost during move from the CSS Transitions specification. (Issue 6097)
- Specified that UA-imposed minimum font sizes apply to the used ‘font-size’ and not to resolution of font-relative lengths. (Issue 5858)
- Clarified how ‘min()’ and ‘max()’ percentages can partially simplify. (Issue 6293)

§ Additions Since Level 3

Changes since CSS Values and Units Level 3:

- Explicitly undefined numeric precision/range.
- Added rules for interpolation per value type, and their clarified computed values.
- Updated interpolation of colors to reference [CSS-COLOR-4].

Additions since CSS Values and Units Level 3:

- Added the ‘mix()’ notation for interpolation.
- Defined the <dashed-ident> type.
- Defined the <ratio> type.
- Added ‘src()’ to the <url> type.
- Added the ‘vi’, ‘vb’, ‘ic’, ‘cap’, ‘lh’ and ‘rlh’ length units.
- Added the ‘svh’, ‘svw’, ‘svi’, ‘svb’, ‘svmin’, and ‘svmax’ small viewport-percentage units and ‘dvh’, ‘dvw’, ‘dvi’, ‘dvb’, ‘dvmin’, and ‘dvmax’ dynamic viewport-percentage units.
- Added the ‘x’ alias to ‘dppx’.
- Added ‘min()’, ‘max()’, and ‘clamp()’ comparison functions.
- Added ‘round()’, ‘mod()’, ‘rem()’, ‘sin()’, ‘cos()’, ‘tan()’, ‘asin()’, ‘acos()’, ‘atan()’, ‘atan2()’, ‘pow()’, ‘sqrt()’, ‘hypot()’, ‘log()’, ‘exp()’, ‘abs()’, ‘sign()’ math functions.
- Added ‘e’, ‘pi’, ‘infinity’, ‘-infinity’, ‘NaN’ constants for use in ‘calc()’.

- Added unit algebra to ‘calc()’, allowing multiplication and division of dimensions.
- A non-integer in a calc() automatically rounds to the nearest integer when used where an <integer> is required.
- Defined serialization of math functions.
- Added a genericized definition of coordinating list property groups, to make it easier to reference the coordinating behavior of the ‘background’ properties.

§ Security Considerations

This specification presents no new security considerations.

This specification defines the ‘url()’ and ‘src()’ functions (<url>), which allow CSS to make network requests. Depending on what features they are used in, these can potentially expose whether or not the user has access to resources on a network, and expose information about their contents (such as the rules within a style sheet, the size of an image, the metrics of a font). They can also allow exfiltrating data via URL.

§ Privacy Considerations

This specification defines units that expose the user’s screen size (the viewport-percentage lengths), default font size, and potentially some information about which fonts are available on the user’s system (the font-relative lengths).

This specification defines the ‘url()’ and ‘src()’ functions (<url>), which allow CSS to make network requests. Depending on what features they are used in, these can potentially expose whether or not the user has access to resources on a network, and expose information about their contents (such as the rules within a style sheet, the size of an image, the metrics of a font). They can also allow exfiltrating data via URL.

§ 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 42
>
> This is an example of an informative example.

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

> Note, this is an informative note.

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

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

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

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

authoring tool
: A UA that writes a style sheet.

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

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

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

§ Partial implementations

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

§ Implementations of Unstable and Proprietary Features

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

§ Non-experimental implementations

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

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

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

public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

<zero>, in § 5.3

§ Terms defined by reference

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

- animation
- animation-iteration-count
- animation-name
- animation-timing-function

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

- margin
- margin-bottom
- margin-left
- margin-right
- margin-top
- padding
- padding-bottom
- padding-left
- padding-right
- padding-top

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

- orphans

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

- @import
- actual value
- computed value
- inherit
- initial
- shorthand property
- specified value
- unset
- used value

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

- <color>
- opacity
- rgba()

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

- @color-profile
- hsl()

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

- @supports
- supports(conditionText)

[css-counter-styles-3] defines the following terms:

- disc

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

- containing block
- initial containing block

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

- <easing-function>
- ease-in
- ease-out
- easing function
- timing function

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

- font
- font-family
- font-size

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

- font-size

[css-grid-2] defines the following terms:

- <flex>
- fr

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

- image-resolution

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

- normal
- vertical-align

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

- clip

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

- max-lines

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

- page area

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

- bottom
- left
- right
- top

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

- block-step-size

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

- rect()

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

- auto
- box-sizing
- height
- max-height
- max-width
- min-height
- min-width
- width

[CSS-SYNTAX-3] defines the following terms:

- <delim-token>
- <dimension-token>
- <function-token>
- <ident-token>
- <number-token>
- <percentage-token>
- <string-token>
- <url-token>
- <whitespace-token>
- component value
- consume a url token
- simple block
- whitespace

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

- center
- letter-spacing
- tab-size
- text-align
- text-indent
- word-spacing

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

- text-decoration

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

- transform-origin

[css-typed-om-1] defines the following terms:

- add two types
- internal representation
- invert a type
- match
- multiply two types
- percent hint
- type

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

- default
- outline-color

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

- attr()

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

- toggle()

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

- custom property
- var()

[css-writing-modes-4] defines the following terms:

- block axis
- inline axis
- text-orientation
- upright
- vertical-lr
- vertical-rl
- writing mode
- writing-mode

[CSS21] defines the following terms:

- <border-width>
- border-collapse
- border-spacing
- line-height
- ua

[CSS3-BACKGROUND] defines the following terms:

- background
- background-attachment
- background-image
- background-position
- border-bottom-width
- border-color
- border-left-width
- border-right-width
- border-top-width
- border-width
- box-shadow
- center

[CSS3-IMAGES] defines the following terms:

- <image>
- linear-gradient()

[CSSOM] defines the following terms:

- CSSStyleSheet
- location
- origin-clean flag
- owner node
- stylesheet base url

[DOM] defines the following terms:

- node tree
- quirks mode

[FETCH] defines the following terms:

- RequestDestination
- client
- credentials mode
- destination
- fetch
- initiator type
- mode
- origin
- processresponseconsumebody
- referrer
- request
- response
- url
- use-url-credentials flag

[HTML] defines the following terms:

- api base url
- base
- origin
- pushState(data, unused)
- relevant settings object

[INFRA] defines the following terms:

- ascii case-insensitive
- concatenate
- for each
- identical to
- string

[mediaqueries-5] defines the following terms:

- continuous media
- media query
- paged media

[URL] defines the following terms:

- url
- url parser

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

- discrete
- not animatable

§ References

§ Normative References

[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-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-COLOR-4]

Tab Atkins Jr.; Chris Lilley; Lea Verou. *CSS Color Module Level 4*. 5 July 2022. CR. URL: https://www.w3.org/TR/css-color-4/

[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-COUNTER-STYLES-3]

Tab Atkins Jr.. *CSS Counter Styles Level 3*. 27 July 2021. CR. URL: https://www.w3.org/TR/css-counter-styles-3/

[CSS-DISPLAY-3]

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

[CSS-EASING-1]

Brian Birtles; et al. *CSS Easing Functions Level 1*. 1 April 2021. CR. URL: https://www.w3.org/TR/css-easing-1/

[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/

[CSS-GRID-2]

Tab Atkins Jr.; Elika Etemad; Rossen Atanassov. *CSS Grid Layout Module Level 2*. 18 December 2020. CR. URL: https://www.w3.org/TR/css-grid-2/

[CSS-IMAGES-4]

Tab Atkins Jr.; Elika Etemad; Lea Verou. *CSS Image Values and Replaced Content Module Level 4*. 13 April 2017. WD. URL: https://www.w3.org/TR/css-images-4/

[CSS-INLINE-3]

Dave Cramer; Elika Etemad; Steve Zilles. *CSS Inline Layout Module Level 3*. 27 August 2020. WD. URL: https://www.w3.org/TR/css-inline-3/

[CSS-MASKING-1]

Dirk Schulze; Brian Birtles; Tab Atkins Jr.. *CSS Masking Module Level 1*. 5 August 2021. CR. URL: https://www.w3.org/TR/css-masking-1/

[CSS-PAGE-3]

Elika Etemad; Simon Sapin. *CSS Paged Media Module Level 3*. 18 October 2018. WD. URL: https://www.w3.org/TR/css-page-3/

[CSS-POSITION-3]

Elika Etemad; Tab Atkins Jr.. *CSS Positioned Layout Module Level 3*. 1 September 2022. WD. URL: https://www.w3.org/TR/css-position-3/

[CSS-SHAPES-1]

Vincent Hardy; Rossen Atanassov; Alan Stearns. *CSS Shapes Module Level 1*. 20 March 2014. CR. URL: https://www.w3.org/TR/css-shapes-1/

[CSS-SIZING-3]

Tab Atkins Jr.; Elika Etemad. *CSS Box Sizing Module Level 3*. 17 December 2021. WD. URL: https://www.w3.org/TR/css-sizing-3/

[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-TEXT-3]

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

[CSS-TYPED-OM-1]

Shane Stephens; Tab Atkins Jr.; Naina Raisinghani. *CSS Typed OM Level 1*. 10 April 2018. WD. URL: https://www.w3.org/TR/css-typed-om-1/

[CSS-UI-3]

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

[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-5]

CSS Values & Units Module Level 5 URL: https://www.w3.org/TR/css-values-5/

[CSS-VARIABLES-1]

Tab Atkins Jr.. *CSS Custom Properties for Cascading Variables Module Level 1*. 16 June 2022. CR. URL: https://www.w3.org/TR/css-variables-1/

[CSS-WRITING-MODES-4]

Elika Etemad; Koji Ishii. *CSS Writing Modes Level 4*. 30 July 2019. CR. URL: https://www.w3.org/TR/css-writing-modes-4/

[CSS21]

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

[CSS3-BACKGROUND]

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/

[CSS3-FONTS]