*:96 Internet application layer protocols and standards

Compendium 9: Allowed during the exam

Last revision: 24 Aug 2004

 [p. ??]

Cascading Style Sheets, level 2 revision 1 CSS 2.1 Specification

W3C Working Draft 15 September 2003

This version:
http://www.w3.org/TR/2003/WD-CSS21-20030915 [p. ??]
Latest version:
http://www.w3.org/TR/CSS21 [p. ??]
Previous version:
http://www.w3.org/TR/2003/WD-CSS21-20030128 [p. ??]
Editors:
Bert Bos [p. ??] <bert @w3.org>
Tantek Çelik [p. ??] <tantekc @microsoft.com>
Ian Hickson [p. ??] <ian @hixie.ch>
Håkon Wium Lie [p. ??] <howcome @opera.com>

This document is also available in these non-normative formats: plain text [p. ??] , gzip'ed tar file [p. ??] , zip file [p. ??] , gzip'ed PostScript [p. ??] , PDF [p. ??] . See also **translations** [p. ??] .

Abstract

This specification defines Cascading Style Sheets, level 2 revision 1 (CSS 2.1). CSS 2.1 is a style sheet language that allows authors and users to attach style (e.g., fonts and spacing) to structured documents (e.g., HTML documents and XML applications). By separating the presentation style of documents from the content of documents, CSS 2.1 simplifies Web authoring and site maintenance.

CSS 2.1 builds on CSS2 [CSS2] which builds on CSS1 [CSS1]. It supports media-specific style sheets so that authors may tailor the presentation of their documents to visual browsers, aural devices, printers, braille devices, handheld devices, etc. It also supports content positioning, table layout, features for internationalization and some properties related to user interface.

CSS 2.1 corrects a few errors in CSS2 (the most important being a new definition of the height/width of absolutely positioned elements, more influence for HTML's "style" attribute and a new calculation of the 'clip' property), and adds a few highly requested features which have already been widely implemented. But most of all CSS 2.1 represents a "snapshot" of CSS usage: it consists of all CSS features that are implemented interoperably at the date of publication of the Recommendation.

Status of this document

This is a W3C Last Call Working Draft [p. ??] . "Last call" means that the working group believes that this specification is ready and therefore wishes this to be the last call for comments. If the feedback is positive, the working group plans to submit it for consideration as a W3C Candidate Recommendation [p. ??] . Comments can be sent until **10 October 2003.**

This document is produced by the CSS working group [p. ??] (part of the Style Activity [p. ??] , see summary [p. ??]).

The (archived [p. ??]) public mailing list www-style@w3.org [p. ??] (see instructions [p. ??]) is preferred for discussion of this and other drafts in the Style area. When commenting on this draft, please put the text "CSS21" in the subject, preferably like this: "[CSS21] <summary of comment>"

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index [p. ??] at http://www.w3.org/TR/. [p. ??] It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than "work in progress."

For this specification to exit the CR stage, the following conditions shall be met:

1. There must be at least two interoperable implementations implementing 'all' the features. An implementation can implement a superset of the features and claim conformance to the profile. For the purposes of this criterion, we define the following terms:
   feature

   An individual test case in the test suite.
   interoperable

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

implementation

a user agent which:

1. implements the feature.
2. is available (i.e. publicly downloadable or available through some other public point of sale mechanism). This is the "show me" requirement.
3. is shipping (i.e. development, private or unofficial versions are insufficient).
4. is not experimental (i.e. is intended for a wide audience and could be used on a daily basis.)

2. A minimum of sixth months of the CR period must have elapsed. This is to ensure that enough time is given for any remaining major errors to be caught.
3. Features may/will be dropped if two or more interoperable implementations are not found by the end of the CR period.
4. Features may/will also be dropped if adequate/sufficient (by judgment of CSS WG) tests have not been produced for those feature(s) by the end of the CR period.

Patent disclosures relevant to CSS may be found on the Working Group's public patent disclosure page. [p. ??]

Quick Table of Contents

Full Table of Contents

1 About the CSS 2.1 Specification

Contents

1.1 CSS 2.1 vs CSS 2

The CSS community has gained significant experience with the CSS2 specification since it became a recommendation in 1998. Errors in the CSS2 specification have subsequently been corrected via the publication of various errata, but there has not yet been an opportunity for the specification to be changed based on experience gained.

While many of these issues will be addressed by the upcoming CSS3 specifications, the current state of affairs hinders the implementation and interoperability of CSS2. The CSS 2.1 specification attempts to address this situation by:

- Maintaining compatibility with those portions of CSS2 that are widely accepted and implemented.
- Incorporating all published CSS2 errata.
- Where implementations overwhelmingly differ from the CSS2 specification, modifying the specification to be in accordance with generally accepted practice.
- Removing all CSS2 features which, by virtue of not having been implemented, have been rejected by the CSS community.
- Removing CSS2 features that will be obsoleted by CSS3, thus encouraging adoption of the proposed CSS3 features in their place.

- Adding a (very) small number of new property values, [p. 297] when implementation experience has shown that they are needed for implementing CSS2.

Thus, while it is not the case that a CSS2 stylesheet is necessarily forwards-compatible with CSS 2.1, it is the case that a stylesheet restricting itself to CSS 2.1 features is more likely to find a compliant user agent today and to preserve forwards compatibility in the future. While breaking forward compatibility is not desirable, we believe the advantages to the revisions in CSS 2.1 are worthwhile.

1.2 Reading the specification

This specification has been written with two types of readers in mind: CSS authors and CSS implementors. We hope the specification will provide authors with the tools they need to write efficient, attractive, and accessible documents, without overexposing them to CSS's implementation details. Implementors, however, should find all they need to build conforming user agents [p. 34] . The specification begins with a general presentation of CSS and becomes more and more technical and specific towards the end. For quick access to information, a general table of contents, specific tables of contents at the beginning of each section, and an index provide easy navigation, in both the electronic and printed versions.

The specification has been written with two modes of presentation in mind: electronic and printed. Although the two presentations will no doubt be similar, readers will find some differences. For example, links will not work in the printed version (obviously), and page numbers will not appear in the electronic version. In case of a discrepancy, the electronic version is considered the authoritative version of the document.

1.3 How the specification is organized

The specification is organized into the following sections:

Section 2: An introduction to CSS2.1
The introduction includes a brief tutorial on CSS2.1 and a discussion of design principles behind CSS2.1.

Sections 3 - 20: CSS 2.1 reference manual.
The bulk of the reference manual consists of the CSS 2.1 language reference. This reference defines what may go into a CSS 2.1 style sheet (syntax, properties, property values) and how user agents must interpret these style sheets in order to claim conformance [p. 34] .

Appendixes:
Appendixes contain information about aural properties [p. 273] (non-normative), a sample style sheet for HTML 4.0 [p. 293] , changes from CSS2 [p. 295] , the grammar of CSS 2.1 [p. 317] , a list of normative and informative references [p. 323] , and two indexes: one for properties [p. 327] and one general index [p. 335] .

1.4 Conventions

1.4.1 Document language elements and attributes

- CSS property, descriptor, and pseudo-class names are delimited by single quotes.
- CSS values are delimited by single quotes.
- Document language element names are in uppercase letters.
- Document language attribute names are in lowercase letters and delimited by double quotes.

1.4.2 CSS property definitions

Each CSS property definition begins with a summary of key information that resembles the following:

'property-name'

Value:	legal values & syntax
Initial:	initial value
Applies to:	elements this property applies to
Inherited:	whether the property is inherited
Percentages:	how percentage values are interpreted
Media:	which media groups the property applies to
Computed value:	how to compute the computed value

Value

This part specifies the set of valid values for the property whose name is 'property-name'. Value types may be designated in several ways:

1. keyword values (e.g., auto, disc, etc.)
2. basic data types, which appear between "<" and ">" (e.g., <length>, <percentage>, etc.). In the electronic version of the document, each instance of a basic data type links to its definition.
3. types that have the same range of values as a property bearing the same name (e.g., <'border-width'> <'background-attachment'>, etc.). In this case, the type name is the property name (complete with quotes) between "<" and ">" (e.g., <'border-width'>). Such a type does **not** include the value 'inherit'. In the electronic version of the document, each instance of this type of non-terminal links to the corresponding property definition.
4. non-terminals that do not share the same name as a property. In this case, the non-terminal name appears between "<" and ">", as in <border-width>. Notice the distinction between <border-width> and <'border-width'>; the latter is defined in terms of the former. The definition of a non-terminal is located near its first appearance in the specification. In the electronic version of the document, each instance of this type of value links to the corresponding value definition.

Other words in these definitions are keywords that must appear literally, without quotes (e.g., red). The slash (/) and the comma (,) must also appear literally.

Values may be arranged as follows:

- Several juxtaposed words mean that all of them must occur, in the given order.
- A bar (|) separates two or more alternatives: exactly one of them must occur.
- A double bar (||) separates two or more options: one or more of them must occur, in any order.
- Brackets ([ ]) are for grouping.

Juxtaposition 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
[ a b ] | [ c || [ d e ]]
```

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.
- A 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 following examples illustrate different value types:

Value: N | NW | NE
Value: [<length> | thick | thin]{1,4}
Value: [<family-name> ,]* <family-name>
Value: <uri>? <color> [/ <color>]?
Value: <uri> || <color>

Value types are specified in terms of tokens, as described in Appendix G.2 [p. 319] . As the grammar allows spaces between tokens in the components of the `expr` production, spaces may appear between tokens in values.

Note: In many cases, spaces will in fact be *required* between tokens in order to distinguish them from each other. For example, the value '1em2em' would be parsed as a single `DIMEN` 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'.

Initial

This part specifies the property's initial value. If the property is inherited, this is the value that is given to the root element of the document tree [p. 33] . Please consult the section on the cascade [p. 79] for information about the interaction between style sheet-specified, inherited, and initial values.

Applies to

This part lists the elements to which the property applies. All elements are considered to have all properties, but some properties have no rendering effect on some types of elements. For example, 'border-spacing' only affects table elements.

Inherited

This part indicates whether the value of the property is inherited from an ancestor element. Please consult the section on the cascade [p. 79] for information about the interaction between style sheet-specified, inherited, and initial values.

Percentage values

This part indicates how percentages should be interpreted, if they occur in the value of the property. If "N/A" appears here, it means that the property does not accept percentages as values.

Media groups

This part indicates the media groups [p. 89] to which the property applies. Information about media groups is non-normative.

Computed value

This part describes the computed value for the property. See the section on computed values [p. 80] for how this definition is used.

1.4.3 Shorthand properties

Some properties are *shorthand properties*, meaning that they allow authors to specify the values of several properties with a single property.

For instance, the 'font' property is a shorthand property for setting 'font-style', 'font-variant', 'font-weight', 'font-size', 'line-height', and 'font-family' all at once.

When values are omitted from a shorthand form, each "missing" property is assigned its initial value (see the section on the cascade [p. 79]).

Example(s):

The multiple style rules of this example:


```
h1 {
  font-weight: bold;
  font-size: 12pt;
  line-height: 14pt;
  font-family: Helvetica;
  font-variant: normal;
  font-style: normal;
}
```

may be rewritten with a single shorthand property:

```
h1 { font: bold 12pt/14pt Helvetica }
```

In this example, 'font-variant', and 'font-style' take their initial values.

1.4.4 Notes and examples

All examples that illustrate illegal usage are clearly marked as "ILLEGAL EXAMPLE".

All HTML examples conform to the HTML 4.0 strict DTD (defined in [HTML40]) unless otherwise indicated by a document type declaration.

All notes are informative only.

Examples and notes are marked within the source HTML for the specification and CSS1 user agents will render them specially.

1.4.5 Images and long descriptions

Most images in the electronic version of this specification are accompanied by "long descriptions" of what they represent. A link to the long description is denoted by a "[D]" to the right of the image.

Images and long descriptions are informative only.

1.5 Acknowledgments

CSS 2.1 is based on CSS2. See the acknowledgments section of CSS2 [p. ??] for the people that contributed to CSS2.

We would like to thank the following people who, through their input and feedback on the www-style mailing list, have helped us with the creation of this specification: Andrew Clover, Bernd Mielke, C. Bottelier, Christian Roth, Christoph Päper, Claus Färber, Coises, Craig Saila, Darren Ferguson, Dylan Schiemann, Etan Wexler, George Lund, James Craig, Jan Eirik Olufsen, Jan Roland Eriksson, Joris Huizer, Joshua Prowse, Kai Lahmann, Kevin Smith, Lachlan Cannon, Lars Knoll, Lars Knoll, Lauri Raittila, Mark Gallagher, Michael Day, Peter Sheerin, Peter Sheerin, Rijk van Geijtenbeek, Robin Berjon, Scott Montgomery, Shelby Moore, Stuart Ballard, Tom Gilder, Vadim Plessky, and the Open eBook Publication Structure Working Group Editors. We would also like to thank Glenn Adams and Susan Lesch, who helped proofread this document.

In addition, we would like to extend special thanks to fantasai, Ada Chan, and Boris Zbarsky, who have both contributed significant time to CSS2.1.

1.6 Copyright Notice

About the CSS 2.1 Specification

2 Introduction to CSS 2.1

Contents

2.1 A brief CSS 2.1 tutorial for HTML

In this tutorial, we show how easy it can be to design simple style sheets. For this tutorial, you will need to know a little HTML (see [HTML40]) and some basic desktop publishing terminology.

We begin with a small HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach's home page</TITLE>
  </HEAD>
  <BODY>
    <H1>Bach's home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

To set the text color of the H1 elements to red, you can write the following CSS rules:

```
h1 { color: red }
```

A CSS rule consists of two main parts: selector [p. 59] ('h1') and declaration ('color: red'). In HTML, element names are case-insensitive so 'h1' works just as well as 'H1'. The declaration has two parts: property ('color') and value ('red'). While the example above tries to influence only one of the properties needed for rendering an HTML document, it qualifies as a style sheet on its own. Combined with other style sheets (one fundamental feature of CSS is that style sheets are combined) it will determine the final presentation of the document.

The HTML 4.0 specification defines how style sheet rules may be specified for HTML documents: either within the HTML document, or via an external style sheet. To put the style sheet into the document, use the STYLE element:


```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach's home page</TITLE>
  <STYLE type="text/css">
    h1 { color: red }
  </STYLE>
  </HEAD>
  <BODY>
    <H1>Bach's home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

For maximum flexibility, we recommend that authors specify external style sheets; they may be changed without modifying the source HTML document, and they may be shared among several documents. To link to an external style sheet, you can use the LINK element:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach's home page</TITLE>
  <LINK rel="stylesheet" href="bach.css" type="text/css">
  </HEAD>
  <BODY>
    <H1>Bach's home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The LINK element specifies:

- the type of link: to a "stylesheet".
- the location of the style sheet via the "href" attribute.
- the type of style sheet being linked: "text/css".

To show the close relationship between a style sheet and the structured markup, we continue to use the STYLE element in this tutorial. Let's add more colors:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach's home page</TITLE>
  <STYLE type="text/css">
    body { color: black; background: white }
    h1 { color: red; background: white }
  </STYLE>
  </HEAD>
  <BODY>
    <H1>Bach's home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The style sheet now contains four rules: the first two set the color and background of the BODY element (it's a good idea to set the text color and background color together), while the last two set the color and the background of the H1 element. Since no color has been specified for the P element, it will inherit the color from its parent element, namely BODY. The H1 element is also a child element of BODY but the second rule overrides the inherited value. In CSS there are often such conflicts between different values, and this specification describes how to resolve them.

CSS 2.1 has more than 90 properties, including 'color'. Let's look at some of the others:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
  <TITLE>Bach's home page</TITLE>
  <STYLE type="text/css">
    body {
      font-family: "Gill Sans", sans-serif;
      font-size: 12pt;
      margin: 3em;
    }
  </STYLE>
  </HEAD>
  <BODY>
    <H1>Bach's home page</H1>
    <P>Johann Sebastian Bach was a prolific composer.
  </BODY>
</HTML>
```

The first thing to notice is that several declarations are grouped within a block enclosed by curly braces ({...}), and separated by semicolons, though the last declaration may also be followed by a semicolon.

The first declaration on the BODY element sets the font family to "Gill Sans". If that font isn't available, the user agent (often referred to as a "browser") will use the 'sans-serif' font family which is one of five generic font families which all users agents know. Child elements of BODY will inherit the value of the 'font-family' property.

The second declaration sets the font size of the BODY element to 12 points. The "point" unit is commonly used in print-based typography to indicate font sizes and other length values. It's an example of an absolute unit which does not scale relative to the environment.

The third declaration uses a relative unit which scales with regard to its surroundings. The "em" unit refers to the font size of the element. In this case the result is that the margins around the BODY element are three times wider than the font size.

2.2 A brief CSS 2.1 tutorial for XML

CSS can be used with any structured document format, for example with applications of the eXtensible Markup Language [XML10]. In fact, XML depends more on style sheets than HTML, since authors can make up their own elements that user agents don't know how to display.

Here is a simple XML fragment:

```
<ARTICLE>
  <HEADLINE>Fredrick the Great meets Bach</HEADLINE>
  <AUTHOR>Johann Nikolaus Forkel</AUTHOR>
  <PARA>
    One evening, just as he was getting his
    <INSTRUMENT>flute</INSTRUMENT> ready and his
    musicians were assembled, an officer brought him a list of
    the strangers who had arrived.
  </PARA>
</ARTICLE>
```

To display this fragment in a document-like fashion, we must first declare which elements are inline-level (i.e., do not cause line breaks) and which are block-level (i.e., cause line breaks).

```
INSTRUMENT { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
```

The first rule declares INSTRUMENT to be inline and the second rule, with its comma-separated list of selectors, declares all the other elements to be block-level. Element names in XML are case-sensitive, so a selector written in lowercase (e.g. 'instrument') is different from uppercase (e.g. 'INSTRUMENT').

One way of linking a style sheet to an XML document is to use a processing instruction:

```
<?xml-stylesheet type="text/css" href="bach.css"?>
<ARTICLE>
  <HEADLINE>Fredrick the Great meets Bach</HEADLINE>
  <AUTHOR>Johann Nikolaus Forkel</AUTHOR>
  <PARA>
    One evening, just as he was getting his
    <INSTRUMENT>flute</INSTRUMENT> ready and his
    musicians were assembled, an officer brought him a list of
    the strangers who had arrived.
  </PARA>
</ARTICLE>
```

A visual user agent could format the above example as:

Fredrick the Great meets Bach
Johann Nikolaus Forkel
One evening, just as he was getting his flute ready and his musicians were assembled, an officer brought him a list of the strangers who had arrived.

Notice that the word "flute" remains within the paragraph since it is the content of the inline element INSTRUMENT.

Still, the text isn't formatted the way you would expect. For example, the headline font size should be larger than then the rest of the text, and you may want to display the author's name in italic:

```
INSTRUMENT { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
HEADLINE { font-size: 1.3em }
AUTHOR { font-style: italic }
ARTICLE, HEADLINE, AUTHOR, PARA { margin: 0.5em }
```

A visual user agent could format the above example as:

Fredrick the Great meets Bach

Johann Nikolaus Forkel

One evening, just as he was getting his flute ready and his musicians were assembled, an officer brought him a list of the strangers who had arrived.

Adding more rules to the style sheet will allow you to further describe the presentation of the document.

2.3 The CSS 2.1 processing model

This section presents one possible model of how user agents that support CSS work. This is only a conceptual model; real implementations may vary.

In this model, a user agent processes a source by going through the following steps:

1. Parse the source document and create a document tree [p. 33] .
2. Identify the target media type [p. 87] .
3. Retrieve all style sheets associated with the document that are specified for the target media type [p. 87] .
4. Annotate every element of the document tree by assigning a single value to every property [p. 45] that is applicable to the target media type [p. 87] . Properties are assigned values according to the mechanisms described in the section on cascading and inheritance [p. 79] .

   Part of the calculation of values depends on the formatting algorithm appropriate for the target media type [p. 87] . For example, if the target medium is the screen, user agents apply the visual formatting model [p. 107] .
5. From the annotated document tree, generate a *formatting structure*. Often, the formatting structure closely resembles the document tree, but it may also differ significantly, notably when authors make use of pseudo-elements and generated content. First, the formatting structure need not be "tree-shaped" at all -- the nature of the structure depends on the implementation. Second, the formatting structure may contain more or less information than the document tree. For instance, if an element in the document tree has a value of 'none' for the 'display' property, that element will generate nothing in the formatting structure. A list element, on the other hand, may generate more information in the formatting structure: the list element's content and list style information (e.g., a bullet image).

   Note that the CSS user agent does not alter the document tree during this phase. In particular, content generated due to style sheets is not fed back to the document language processor (e.g., for reparsing).
6. Transfer the formatting structure to the target medium (e.g., print the results, display them on the screen, render them as speech, etc.).

Step 1 lies outside the scope of this specification (see, for example, [DOM]).

Steps 2-5 are addressed by the bulk of this specification.

Step 6 lies outside the scope of this specification.

2.3.1 The canvas

For all media, the term *canvas* describes "the space where the formatting structure is rendered." The canvas is infinite for each dimension of the space, but rendering generally occurs within a finite region of the canvas, established by the user agent according to the target medium. For instance, user agents rendering to a screen generally impose a minimum width and choose an initial width based on the dimensions of the viewport [p. 108] . User agents rendering to a page generally impose width and height constraints. Aural user agents may impose limits in audio space, but not in time.

2.3.2 CSS 2.1 addressing model

CSS 2.1 selectors [p. 59] and properties allow style sheets to refer to the following parts of a document or user agent:

- Elements in the document tree and certain relationships between them (see the section on selectors [p. 59]).
- Attributes of elements in the document tree, and values of those attributes (see

the section on attribute selectors [p. 64]).
- Some parts of element content (see the :first-line [p. 75] and :first-letter [p. 75] pseudo-elements).
- Elements of the document tree when they are in a certain state (see the section on pseudo-classes [p. 69]).
- Some aspects of the canvas [p. 28] where the document will be rendered.
- Some system information (see the section on user interface [p. 259]).

2.4 CSS design principles

CSS 2.1, as CSS2 and CSS1 before it, is based on a set of design principles:

- **Forward and backward compatibility**. CSS 2.1 user agents will be able to understand CSS1 style sheets. CSS1 user agents will be able to read CSS 2.1 style sheets and discard parts they don't understand. Also, user agents with no CSS support will be able to display style-enhanced documents. Of course, the stylistic enhancements made possible by CSS will not be rendered, but all content will be presented.
- **Complementary to structured documents**. Style sheets complement structured documents (e.g., HTML and XML applications), providing stylistic information for the marked-up text. It should be easy to change the style sheet with little or no impact on the markup.
- **Vendor, platform, and device independence**. Style sheets enable documents to remain vendor, platform, and device independent. Style sheets themselves are also vendor and platform independent, but CSS 2.1 allows you to target a style sheet for a group of devices (e.g., printers).
- **Maintainability**. By pointing to style sheets from documents, webmasters can simplify site maintenance and retain consistent look and feel throughout the site. For example, if the organization's background color changes, only one file needs to be changed.
- **Simplicity**. CSS is a simple style language which is human readable and writable. The CSS properties are kept independent of each other to the largest extent possible and there is generally only one way to achieve a certain effect.
- **Network performance**. CSS provides for compact encodings of how to present content. Compared to images or audio files, which are often used by authors to achieve certain rendering effects, style sheets most often decrease the content size. Also, fewer network connections have to be opened which further increases network performance.
- **Flexibility**. CSS can be applied to content in several ways. The key feature is the ability to cascade style information specified in the default (user agent) style sheet, user style sheets, linked style sheets, the document head, and in attributes for the elements forming the document body.

- **Richness**. Providing authors with a rich set of rendering effects increases the richness of the Web as a medium of expression. Designers have been longing for functionality commonly found in desktop publishing and slide-show applications. Some of the requested rendering effects conflict with device independence, but CSS 2.1 goes a long way toward granting designers their requests.
- **Alternative language bindings**. The set of CSS properties described in this specification form a consistent formatting model for visual and aural presentations. This formatting model can be accessed through the CSS language, but bindings to other languages are also possible. For example, a JavaScript program may dynamically change the value of a certain element's 'color' property.
- **Accessibility**. Several CSS features will make the Web more accessible to users with disabilities:
  - Properties to control font appearance allow authors to eliminate inaccessible bit-mapped text images.
  - Positioning properties allow authors to eliminate mark-up tricks (e.g., invisible images) to force layout.
  - The semantics of `!important` rules mean that users with particular presentation requirements can override the author's style sheets.
  - The 'inherit' value for all properties improves cascading generality and allows for easier and more consistent style tuning.
  - Improved media support, including media groups and the braille, embossed, and tty media types, will allow users and authors to tailor pages to those devices.

Note. *For more information about designing accessible documents using CSS and HTML, see [WAI-PAGEAUTH].*

3 Conformance: Requirements and Recommendations

Contents

3.1 Definitions

In this section, we begin the formal specification of CSS 2.1, starting with the contract between authors, users, and implementors.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 (see [RFC2119]). However, for readability, these words do not appear in all uppercase letters in this specification.

At times, this specification recommends good practice for authors and user agents. These recommendations are not normative and conformance with this specification does not depend on their realization. These recommendations contain the expression "We recommend ...", "This specification recommends ...", or some similar wording.

Style sheet
A set of statements that specify presentation of a document.

Style sheets may have three different origins: author [p. 33] , user [p. 33] , and user agent [p. 33] . The interaction of these sources is described in the section on cascading and inheritance [p. 79] .

Valid style sheet
The validity of a style sheet depends on the level of CSS used for the style sheet. All valid CSS1 style sheets are valid CSS 2.1 style sheets, but some changes from CSS1 mean that a few CSS1 style sheets will have slightly different semantics in CSS 2.1. Some features in CSS2 are not part of CSS 2.1, so not all CSS2 style sheets are valid CSS 2.1 style sheets.

A valid CSS 2.1 style sheet must be written according to the grammar of CSS 2.1 [p. 317] . Furthermore, it must contain only at-rules, property names, and property values defined in this specification. An **illegal** (invalid) at-rule, property name, or property value is one that is not valid.

Source document
The document to which one or more style sheets apply. This is encoded in some language that represents the document as a tree of elements [p. 32] . Each element consists of a name that identifies the type of element, optionally a number of attributes [p. 32] , and a (possibly empty) content [p. 32] .

Document language
The encoding language of the source document (e.g., HTML, XHTML or SVG). CSS is used to describe the presentation of document languages and CSS does not change the underlying semantics of the document languages.

Element
(An SGML term, see [ISO8879].) The primary syntactic constructs of the document language. Most CSS style sheet rules use the names of these elements (such as P, TABLE, and OL in HTML) to specify how the elements should be rendered.

Replaced element
An element for which the CSS formatter knows only the intrinsic dimensions. In HTML, IMG and OBJECT elements can be replaced elements. For example, the content of the IMG element is often replaced by the image that the "src" attribute designates.

Intrinsic dimensions
The width and height as defined by the element itself, not imposed by the surroundings. CSS does not define how the intrinsic dimensions are found. In CSS 2.1 it is assumed that all replaced elements, and only replaced elements, come with intrinsic dimensions.

Attribute
A value associated with an element, consisting of a name, and an associated (textual) value.

Content
The content associated with an element in the source document. Some elements have no content, in which case they are called **empty**. The content of an element may include text, and it may include a number of sub-elements, in which case the element is called the **parent** of those sub-elements.

Ignore
This term has three slightly different meanings this specification. First, a CSS parser must follow certain rules when it discovers unknown or illegal syntax in a style sheet. The parser must then ignore certain parts of the style sheets. The exact rules for what parts must be ignored is given in these section: Declarations and properties [p. ??] , Rules for handling parsing errors [p. 46] , Unsupported Values [p. ??] , or may be explained in the text where the term "ignore" appears. Second, a user agent may (and, in some cases must) disregard certain properties or values in the style sheet even if the syntax is legal. For example, table-column-group elements cannot have borders around them, so the border properties must be ignored.

Rendered content
The content of an element after the rendering that applies to it according to the relevant style sheets has been applied. The rendered content of a replaced

element [p. 32] comes from outside the source document. Rendered content may also be alternate text for an element (e.g., the value of the XHTML "alt" attribute), and may include items inserted implicitly or explicitly by the style sheet, such as bullets, numbering, etc.

Document tree
The tree of elements encoded in the source document. Each element in this tree has exactly one parent, with the exception of the **root** element, which has none.

Child
An element A is called the child of element B if and only if B is the parent of A.

Descendant
An element A is called a descendant of an element B, if either (1) A is a child of B, or (2) A is the child of some element C that is a descendant of B.

Ancestor
An element A is called an ancestor of an element B, if and only if B is a descendant of A.

Sibling
An element A is called a sibling of an element B, if and only if B and A share the same parent element. Element A is a preceding sibling if it comes before B in the document tree. Element B is a following sibling if it comes after A in the document tree.

Preceding element
An element A is called a preceding element of an element B, if and only if (1) A is an ancestor of B or (2) A is a preceding sibling of B.

Following element
An element A is called a following element of an element B, if and only if B is a preceding element of A.

Author
An author is a person who writes documents and associated style sheets. An **authoring tool** generates documents and associated style sheets.

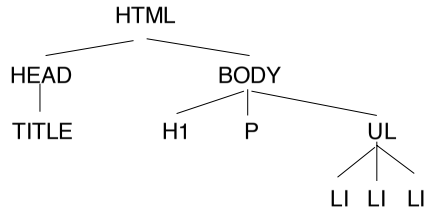
User
A user is a person who interacts with a user agent to view, hear, or otherwise use a document and its associated style sheet. The user may provide a personal style sheet that encodes personal preferences.

User agent (UA)
A user agent is any program that interprets a document written in the document language and applies associated style sheets according to the terms of this specification. A user agent may display a document, read it aloud, cause it to be printed, convert it to another format, etc.
An HTML user agent is one that supports the HTML 2.x, HTML 3.x, or HTML 4.x specifications. A user agent that supports XHTML [XHTML], but not HTML (as listed in the previous sentence) is not considered an HTML user agent for the purpose of conformance with this specification.

Here is an example of a source document written in HTML:


```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <TITLE>My home page</TITLE>
  <BODY>
    <H1>My home page</H1>
    <P>Welcome to my home page! Let me tell you about my favorite
                composers:
    <UL>
      <LI> Elvis Costello
      <LI> Johannes Brahms
      <LI> Georges Brassens
    </UL>
  </BODY>
</HTML>
```

This results in the following tree:



According to the definition of HTML 4.0, HEAD elements will be inferred during parsing and become part of the document tree even if the "head" tags are not in the document source. Similarly, the parser knows where the P and LI elements end, even though there are no </p> and tags in the source.

Documents written in XHTML (and other XML-based languages) behave differently: there are no inferred elements and all elements must have end tags.

3.2 Conformance

This section defines conformance with the CSS 2.1 specification only. There may be other levels of CSS in the future that may require a user agent to implement a different set of features in order to conform.

In general, the following points must be observed by a user agent claiming conformance to this specification:

1. It must support one or more of the CSS 2.1 media types [p. 87] .
2. For each source document, it must attempt to retrieve all associated style sheets that are appropriate for the supported media types. If it cannot retrieve all associated style sheets (for instance, because of network errors), it must display the document using those it can retrieve.
3. It must parse the style sheets according to this specification. In particular, it must recognize all at-rules, blocks, declarations, and selectors (see the grammar of CSS 2.1 [p. 317]). If a user agent encounters a property that

applies for a supported media type, the user agent must parse the value according to the property definition. This means that the user agent must accept all valid values and must ignore declarations with invalid values. User agents must ignore rules that apply to unsupported media types [p. 87] .

4. For each element in a document tree [p. 33] , it must assign a value for every applicable property according to the property's definition and the rules of cascading and inheritance [p. 79] .
5. If the source document comes with alternate style sheet sets (such as with the "alternate" keyword in HTML 4.0 [HTML40]), the UA must allow the user to select which style sheet set the UA should apply.

Not every user agent must observe every point, however:

- An application that reads style sheets without rendering any content (e.g., a CSS 2.1 validator) must respect points 1-3.
- An authoring tool is only required to output valid style sheets [p. 31]
- A user agent that *renders* a document with associated style sheets must respect points 1-5 and render the document according to the media-specific requirements set forth in this specification. Values [p. 80] may be approximated when required by the user agent.

The inability of a user agent to implement part of this specification due to the limitations of a particular device (e.g., a user agent cannot render colors on a monochrome monitor or page) does not imply non-conformance.

UAs must allow users to specify a file that contains the user style sheet. UAs that run on devices without any means of writing or specifying files are exempted from this requirement. Additionally, UAs may offer other means to specify user preferences, for example through a GUI.

CSS2.1 does not define which properties apply to form controls and frames, or how CSS can be used to style them. User agents may apply CSS properties to these elements. Authors are recommended to treat such support as experimental. A future level of CSS may specify this further.

3.3 Error conditions

In general, this document does not specify error handling behavior for user agents (e.g., how they behave when they cannot find a resource designated by a URI).

However, user agents must observe the rules for handling parsing errors [p. 46] .

Since user agents may vary in how they handle error conditions, authors and users must not rely on specific error recovery behavior.

3.4 The text/css content type

CSS style sheets that exist in separate files are sent over the Internet as a sequence of bytes accompanied by encoding information. The structure of the transmission, termed a **message entity,** is defined by RFC 2045 and RFC 2068 (see [RFC2045] and [RFC2068]). A message entity with a content type of "text/css" represents an independent CSS document. The "text/css" content type has been registered by RFC 2318 ([RFC2318]).

4 Syntax and basic data types

Contents

4.1 Syntax

This section describes a grammar (and *forward-compatible parsing* rules) common to any version of CSS (including CSS 2.1). Future versions of CSS will adhere to this core syntax, although they may add additional syntactic constraints.

These descriptions are normative. They are also complemented by the normative grammar rules presented in Appendix G [p. 317] .

4.1.1 Tokenization

All levels of CSS — level 1, level 2, and any future levels — use the same core syntax. This allows UAs to parse (though not completely understand) style sheets written in levels of CSS that didn't exist at the time the UAs were created. Designers can use this feature to create style sheets that work with older user agents, while

also exercising the possibilities of the latest levels of CSS.

At the lexical level, CSS style sheets consist of a sequence of tokens. The list of tokens for CSS 2.1 is as follows. The definitions use Lex-style regular expressions. Octal codes refer to ISO 10646 ([ISO10646]). As in Lex, in case of multiple matches, the longest match determines the token.

Token	Definition
IDENT	*{ident}*
ATKEYWORD	@*{ident}*
STRING	*{string}*
HASH	#*{name}*
NUMBER	*{num}*
PERCENTAGE	*{num}*%
DIMENSION	*{num}{ident}*
URI	url\(*{w}{string}{w}*\) |url\(*{w}*([!#$%&*-~]|*{nonascii}*|*{escape}*)*{w}*\)
UNICODE-RANGE	U\+[0-9A-F?]{1,6}(-[0-9A-F]{1,6})?
CDO	<!--
CDC	-->
;	;
{	\{
}	\}
(	\(
)	\)
[	\[
]	\]
S	[\t\r\n\f]+
COMMENT	\/*[^*]**+([^/*][^*]**+)*\/
FUNCTION	*{ident}*\(
INCLUDES	~=
DASHMATCH	|=
DELIM	*any other character not matched by the above rules, and neither a single nor a double quote*

The macros in curly braces ({}) above are defined as follows:

Macro	Definition
ident	[-]?*{nmstart}{nmchar}**
name	*{nmchar}*+
nmstart	[_a-zA-Z]|*{nonascii}*|*{escape}*
nonascii	[^\0-\177]
unicode	\\[0-9a-f]{1,6}(\r\n|[\n\r\t\f])?
escape	*{unicode}*|\\[-~\200-\4177777]
nmchar	[_a-zA-Z0-9-]|*{nonascii}*|*{escape}*
num	[0-9]+|[0-9]*\.[0-9]+
string	*{string1}*|*{string2}*
string1	\"([\t !#$%&(-~]|\\{nl}|\'|*{nonascii}*|*{escape}*)*\"
string2	\'([\t !#$%&(-~]|\\{nl}|\"|*{nonascii}*|*{escape}*)*\'
nl	\n|\r\n|\r|\f
w	[\t\r\n\f]*

Below is the core syntax for CSS. The sections that follow describe how to use it. Appendix G [p. 317] describes a more restrictive grammar that is closer to the CSS level 2 language.

```
stylesheet  : [ CDO | CDC | S | statement ]*;
statement   : ruleset | at-rule;
at-rule     : ATKEYWORD S* any* [ block | ';' S* ];
block       : '{' S* [ any | block | ATKEYWORD S* | ';' S* ]* '}' S*;
ruleset     : selector? '{' S* declaration? [ ';' S* declaration? ]* '}' S*;
selector    : any+;
declaration : DELIM? property S* ':' S* value;
property    : IDENT;
value       : [ any | block | ATKEYWORD S* ]+;
any         : [ IDENT | NUMBER | PERCENTAGE | DIMENSION | STRING
              | DELIM | URI | HASH | UNICODE-RANGE | INCLUDES
              | DASHMATCH | FUNCTION S* any* ')'
              | '(' S* any* ')' | '[' S* any* ']' ] S*;
```

COMMENT tokens do not occur in the grammar (to keep it readable), but any number of these tokens may appear anywhere between other tokens.

The token S in the grammar above stands for whitespace. Only the characters "space" (Unicode code 32), "tab" (9), "line feed" (10), "carriage return" (13), and "form feed" (12) can occur in whitespace. Other space-like characters, such as "em-space" (8195) and "ideographic space" (12288), are never part of whitespace.

4.1.2 Keywords

Keywords have the form of identifiers. Keywords must not be placed between quotes ("..." or '...'). Thus,

```
red
```

is a keyword, but

```
"red"
```

is not. (It is a string [p. 54] .) Other illegal examples:

Illegal example(s):

```
width: "auto";
border: "none";
background: "red";
```

Vendor-specific extensions

In CSS2.1, identifiers may begin with '-' (dash) or '_' (underscore). Keywords and property names, beginning with - or '_' are reserved for vendor-specific extensions. Such vendor-specific extensions should have one of the following formats:

```
'-' + vendor identifier + '-' + meaningful name
'_' + vendor identifier + '-' + meaningful name
```

Example(s):

For example, if XYZ organization added a property to describe the color of the border on the East side of the display, they might call it -xyz-border-east-color.

Other known examples:

```
-moz-box-sizing
-moz-border-radius
-wap-accesskey
```

An initial dash or underscore is guaranteed never to be used in a property or keyword by any current or future level of CSS. Thus typical CSS implementations may not recognize such properties and may ignore them according to the rules for handling parsing errors [p. 46] . However, because the initial dash or underscore is part of the grammar, CSS2.1 implementers should always be able to use a CSS-conforming parser, whether or not they support any vendor-specific extensions.

Informative Historical Notes

This section is informative.

At the time of writing, the following prefixes are known to exist:

prefix	organization	notes
mso-	Microsoft Corporation	Created before the working group established a naming convention for extensions.
-moz-	The Mozilla Organization	
-o-	Opera Software	
-atsc-	Advanced Television Standards Committee	
-wap-	The WAP Forum	

Vendor/organization specific extensions should be avoided.

4.1.3 Characters and case

The following rules always hold:

- All CSS style sheets are case-insensitive, except for parts that are not under the control of CSS. For example, the case-sensitivity of values of the HTML attributes "id" and "class", of font names, and of URIs lies outside the scope of this specification. Note in particular that element names are case-insensitive in HTML, but case-sensitive in XML.
- In CSS 2.1, *identifiers* (including element names, classes, and IDs in selectors [p. 59]) can contain only the characters [A-Za-z0-9] and ISO 10646 characters 161 and higher, plus the hyphen (-) and the underscore (_); they cannot start with a hyphen or a digit. They can also contain escaped characters and any ISO 10646 character as a numeric code (see next item). For instance, the identifier "B&W?" may be written as "B\&W\?" or "B\26 W\3F".

  Note that Unicode is code-by-code equivalent to ISO 10646 (see [UNICODE] and [ISO10646]).
- In CSS 2.1, a backslash (\) character indicates three types of character escapes.

  First, inside a string [p. 54] , a backslash followed by a newline is ignored (i.e., the string is deemed not to contain either the backslash or the newline).

  Second, it cancels the meaning of special CSS characters. Any character (except a hexadecimal digit) can be escaped with a backslash to remove its special meaning. For example, `"\""` is a string consisting of one double quote. Style sheet preprocessors must not remove these backslashes from a style sheet since that would change the style sheet's meaning.

  Third, backslash escapes allow authors to refer to characters they can't easily put in a document. In this case, the backslash is followed by at most six hexadecimal digits (0..9A..F), which stand for the ISO 10646 ([ISO10646]) character with that number. If a character in the range [0-9a-zA-Z] follows the hexadecimal number, the end of the number needs to be made clear. There are

two ways to do that:

1. with a space (or other whitespace character): "\26 B" ("&B"). In this case, user agents should treat a "CR/LF" pair (13/10) as a single whitespace character.
2. by providing exactly 6 hexadecimal digits: "\000026B" ("&B")

In fact, these two methods may be combined. Only one whitespace character is ignored after a hexadecimal escape. Note that this means that a "real" space after the escape sequence must itself either be escaped or doubled.

- Backslash escapes are always considered to be part of an identifier [p. 42] or a string (i.e., "\7B" is not punctuation, even though "{" is, and "\32" is allowed at the start of a class name, even though "2" is not).

4.1.4 Statements

A CSS style sheet, for any version of CSS, consists of a list of *statements* (see the grammar above). There are two kinds of statements: *at-rules* and *rule sets.* There may be whitespace [p. 40] around the statements.

In this specification, the expressions "immediately before" or "immediately after" mean with no intervening whitespace or comments.

4.1.5 At-rules

At-rules start with an *at-keyword*, an '@' character followed immediately by an identifier [p. 42] (for example, '@import', '@page').

An at-rule consists of everything up to and including the next semicolon (;) or the next block, [p. 44] whichever comes first. A CSS user agent that encounters an unrecognized at-rule must ignore [p. 46] the whole of the at-rule and continue parsing after it.

CSS 2.1 user agents must ignore [p. 46] any '@import' [p. 81] rule that occurs inside a block [p. 44] or that doesn't precede all rule sets.

Illegal example(s):

Assume, for example, that a CSS 2.1 parser encounters this style sheet:

```
@import "subs.css";
h1 { color: blue }
@import "list.css";
```

The second '@import' is illegal according to CSS2.1. The CSS 2.1 parser ignores [p. 46] the whole at-rule, effectively reducing the style sheet to:

```
@import "subs.css";
h1 { color: blue }
```


Illegal example(s):

In the following example, the second '@import' rule is invalid, since it occurs inside a '@media' block [p. 44] .

```
@import "subs.css";
@media print {
  @import "print-main.css";
  body { font-size: 10pt }
}
h1 {color: blue }
```

4.1.6 Blocks

A *block* starts with a left curly brace ({) and ends with the matching right curly brace (}). In between there may be any characters, except that parentheses (( )), brackets ([ ]) and braces ({ }) must always occur in matching pairs and may be nested. Single (') and double quotes (") must also occur in matching pairs, and characters between them are parsed as a string. See Tokenization [p. 37] above for the definition of a string.

Illegal example(s):

Here is an example of a block. Note that the right brace between the double quotes does not match the opening brace of the block, and that the second single quote is an escaped character [p. 42] , and thus doesn't match the first single quote:

```
{ causta: "}" + ({7} * '\'') }
```

Note that the above rule is not valid CSS 2.1, but it is still a block as defined above.

4.1.7 Rule sets, declaration blocks, and selectors

A rule set (also called "rule") consists of a selector followed by a declaration block.

A *declaration-block* (also called a {}-block in the following text) starts with a left curly brace ({) and ends with the matching right curly brace (}). In between there must be a list of zero or more semicolon-separated (;) declarations.

The *selector* (see also the section on selectors [p. 59]) consists of everything up to (but not including) the first left curly brace ({). A selector always goes together with a {}-block. When a user agent can't parse the selector (i.e., it is not valid CSS 2.1), it must ignore [p. 46] the {}-block as well.

CSS 2.1 gives a special meaning to the comma (,) in selectors. However, since it is not known if the comma may acquire other meanings in future versions of CSS, the whole statement should be ignored [p. 46] if there is an error anywhere in the selector, even though the rest of the selector may look reasonable in CSS 2.1.

Illegal example(s):

For example, since the "&" is not a valid token in a CSS 2.1 selector, a CSS 2.1 user agent must ignore [p. 46] the whole second line, and not set the color of H3 to red:

```
h1, h2 {color: green }
h3, h4 & h5 {color: red }
h6 {color: black }
```

Example(s):

Here is a more complex example. The first two pairs of curly braces are inside a string, and do not mark the end of the selector. This is a valid CSS 2.1 statement.

```
p[example="public class foo\
{\
    private int x;\
\
    foo(int x) {\
        this.x = x;\
    }\
\
}"] { color: red }
```

4.1.8 Declarations and properties

A *declaration* is either empty or consists of a property, followed by a colon (:), followed by a value. Around each of these there may be whitespace [p. 40] .

Because of the way selectors work, multiple declarations for the same selector may be organized into semicolon (;) separated groups.

Example(s):

Thus, the following rules:

```
h1 { font-weight: bold }
h1 { font-size: 12px }
h1 { line-height: 14px }
h1 { font-family: Helvetica }
h1 { font-variant: normal }
h1 { font-style: normal }
```

are equivalent to:

```
h1 {
  font-weight: bold;
  font-size: 12px;
  line-height: 14px;
  font-family: Helvetica;
  font-variant: normal;
  font-style: normal
}
```

A property is an identifier [p. 42] . Any character may occur in the value. Parentheses ("( )"), brackets ("[ ]"), braces ("{ }"), single quotes (') and double quotes (") must come in matching pairs, and semicolons not in strings must be escaped [p. 42] .

Parentheses, brackets, and braces may be nested. Inside the quotes, characters are parsed as a string.

The syntax of values is specified separately for each property, but in any case, values are built from identifiers, strings, numbers, lengths, percentages, URIs, and colors.

A user agent must ignore [p. 46] a declaration with an invalid property name or an invalid value. Every CSS 2.1 property has its own syntactic and semantic restrictions on the values it accepts.

Illegal example(s):

For example, assume a CSS 2.1 parser encounters this style sheet:

```
h1 { color: red; font-style: 12pt }  /* Invalid value: 12pt */
p { color: blue;  font-vendor: any;  /* Invalid prop.: font-vendor */
    font-variant: small-caps }
em em { font-style: normal }
```

The second declaration on the first line has an invalid value '12pt'. The second declaration on the second line contains an undefined property 'font-vendor'. The CSS 2.1 parser will ignore [p. 46] these declarations, effectively reducing the style sheet to:

```
h1 { color: red; }
p { color: blue;  font-variant: small-caps }
em em { font-style: normal }
```

4.1.9 Comments

Comments begin with the characters "/*" and end with the characters "*/". They may occur anywhere between tokens, and their contents have no influence on the rendering. Comments may not be nested.

CSS also allows the SGML comment delimiters ("<!--" and "-->") in certain places, but they do not delimit CSS comments. They are permitted so that style rules appearing in an HTML source document (in the STYLE element) may be hidden from pre-HTML 3.2 user agents. See the HTML 4.0 specification ([HTML40]) for more information.

4.2 Rules for handling parsing errors

In some cases, user agents must ignore part of an illegal style sheet. This specification defines *ignore* to mean that the user agent parses the illegal part (in order to find its beginning and end), but otherwise acts as if it had not been there.

To ensure that new properties and new values for existing properties can be added in the future, user agents are required to obey the following rules when they encounter the following scenarios:

- **Unknown properties.** User agents must ignore [p. 46] a declaration [p. 45] with an unknown property. For example, if the style sheet is:

```
h1 { color: red; rotation: 70minutes }
```

the user agent will treat this as if the style sheet had been

```
h1 { color: red }
```

- **Illegal values.** User agents must ignore a declaration with an illegal value. For example:

```
img { float: left }       /* correct CSS 2.1 */
img { float: left here }  /* "here" is not a value of 'float' */
img { background: "red" } /* keywords cannot be quoted */
img { border-width: 3 }   /* a unit must be specified for length values */
```

A CSS 2.1 parser would honor the first rule and ignore [p. 46] the rest, as if the style sheet had been:

```
img { float: left }
img { }
img { }
img { }
```

A user agent conforming to a future CSS specification may accept one or more of the other rules as well.

- **Malformed declarations.** User agents must handle unexpected tokens encountered while parsing a declaration by reading until the end of the declaration, while observing the rules for matching pairs of (), [], {}, "", and '', and correctly handling escapes. For example, a malformed declaration may be missing a property, colon (:) or value. The following are all equivalent:

```
p { color:green }
p { color:green; color }  /* malformed declaration missing ':', value */
p { color:red;   color; color:green }  /* same with expected recovery */
p { color:green; color: } /* malformed declaration missing value */
p { color:red;   color:; color:green } /* same with expected recovery */
p { color:green; color{;color:maroon} } /* unexpected tokens { } */
p { color:red;   color{;color:maroon}; color:green } /* same with recovery */
```

- **Invalid at-keywords.** User agents must ignore [p. 46] an invalid at-keyword together with everything following it, up to and including the next semicolon (;) or block ({...}), whichever comes first. For example, consider the following:

```
@three-dee {
  @background-lighting {
    azimuth: 30deg;
    elevation: 190deg;
  }
  h1 { color: red }
}
h1 { color: blue }
```


The '@three-dee' at-rule is not part of CSS 2.1. Therefore, the whole at-rule (up to, and including, the third right curly brace) is ignored. [p. 46] A CSS 2.1 user agent ignores [p. 46] it, effectively reducing the style sheet to:

```
h1 { color: blue }
```

4.3 Values

4.3.1 Integers and real numbers

Some value types may have integer values (denoted by <integer>) or real number values (denoted by <number>). Real numbers and integers are specified in decimal notation only. An <integer> consists of one or more digits "0" to "9". A <number> can either be an <integer>, or it can be zero or more digits followed by a dot (.) followed by one or more digits. Both integers and real numbers may be preceded by a "-" or "+" to indicate the sign.

Note that many properties that allow an integer or real number as a value actually restrict the value to some range, often to a non-negative value.

4.3.2 Lengths

Lengths refer to horizontal or vertical measurements.

The format of a length value (denoted by <length> in this specification) is a <number> (with or without a decimal point) immediately followed by a unit identifier (e.g., px, em, etc.). After a zero length, the unit identifier is optional.

Some properties allow negative length values, but this may complicate the formatting model and there may be implementation-specific limits. If a negative length value cannot be supported, it should be converted to the nearest value that can be supported.

If a negative length value is set on a property that does not allow negative length values, the declaration is ignored.

There are two types of length units: relative and absolute. *Relative length* units specify a length relative to another length property. Style sheets that use relative units will more easily scale from one medium to another (e.g., from a computer display to a laser printer).

Relative units are:

- **em**: the 'font-size' of the relevant font
- **ex**: the 'x-height' of the relevant font
- **px**: pixels, relative to the viewing device

Example(s):

```
h1 { margin: 0.5em }      /* em */
h1 { margin: 1ex }        /* ex */
p  { font-size: 12px }    /* px */
```

The 'em' unit is equal to the computed value of the 'font-size' property of the element on which it is used. The exception is when 'em' occurs in the value of the 'font-size' property itself, in which case it refers to the font size of the parent element. It may be used for vertical or horizontal measurement. (This unit is also sometimes called the quad-width in typographic texts.)

The 'ex' unit is defined by the font's 'x-height'. 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 don't contain an "x".

Example(s):

The rule:

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

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

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

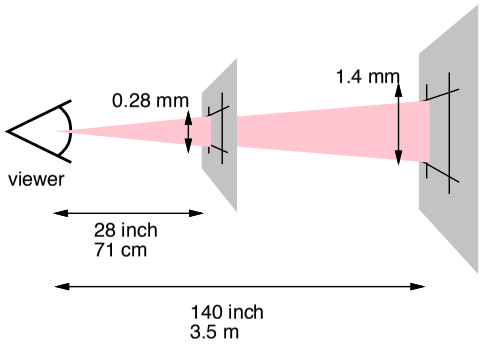
means that the font-size of "h1" elements will be 20% greater than the font size inherited by "h1" elements.

When specified for the root of the document tree [p. 33] (e.g., "HTML" in HTML), 'em' and 'ex' refer to the property's initial value [p. 19] .

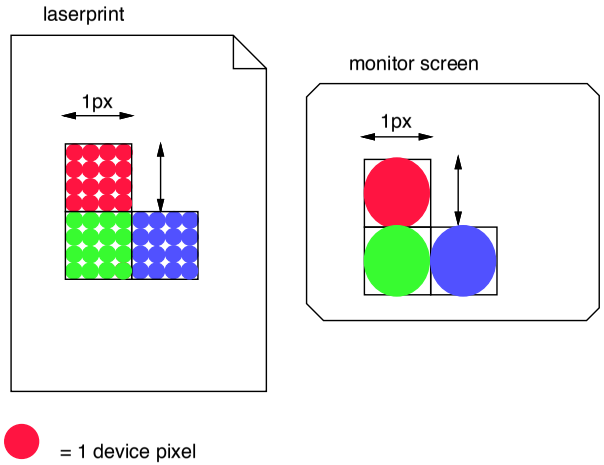
Pixel units are relative to the resolution of the viewing device, i.e., most often a computer display. If the pixel density of the output device is very different from that of a typical computer display, the user agent should rescale pixel values. It is recommended that the *reference pixel* be the visual angle of one pixel on a device with a pixel density of 96dpi and a distance from the reader of an arm's length. For a nominal arm's length of 28 inches, the visual angle is therefore about 0.0213 degrees.

For reading at arm's length, 1px thus corresponds to about 0.26 mm (1/96 inch). When printed on a laser printer, meant for reading at a little less than arm's length (55 cm, 21 inches), 1px is about 0.20 mm. On a 300 dots-per-inch (dpi) printer, that may be rounded up to 3 dots (0.25 mm); on a 600 dpi printer, it can be rounded to 5 dots.

The two images below illustrate the effect of viewing distance on the size of a pixel and the effect of a device's resolution. In the first image, a reading distance of 71 cm (28 inch) results in a px of 0.26 mm, while a reading distance of 3.5 m (12 feet) requires a px of 1.3 mm.



In the second image, an area of 1px by 1px is covered by a single dot in a low-resolution device (a computer screen), while the same area is covered by 16 dots in a higher resolution device (such as a 400 dpi laser printer).



Child elements do not inherit the relative values specified for their parent; they (generally) inherit the computed values [p. 80] .

Example(s):

In the following rules, the computed 'text-indent' value of "h1" elements will be 36px, not 45px, if "h1" is a child of the "body" element.

```
body {
  font-size: 12px;
  text-indent: 3em;  /* i.e., 36px */
}
h1 { font-size: 15px }
```

Absolute length units are only useful when the physical properties of the output medium are known. The absolute units are:

- **in**: inches — 1 inch is equal to 2.54 centimeters.
- **cm**: centimeters
- **mm**: millimeters
- **pt**: points — the points used by CSS 2.1 are equal to 1/72th of an inch.
- **pc**: picas — 1 pica is equal to 12 points.

Example(s):

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

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

4.3.3 Percentages

The format of a percentage value (denoted by <percentage> in this specification) is a <number> immediately followed by '%'.

Percentage values are always relative to another value, for example a length. Each property that allows percentages also defines the value to which the percentage refers. The value may be that of another property for the same element, a property for an ancestor element, or a value of the formatting context (e.g., the width of a containing block [p. 108]). When a percentage value is set for a property of the root [p. 33] element and the percentage is defined as referring to the inherited value of some property, the resultant value is the percentage times the initial value [p. 19] of that property.

Example(s):

Since child elements (generally) inherit the computed values [p. 80] of their parent, in the following example, the children of the P element will inherit a value of 12px for 'line-height', not the percentage value (120%):

```
p { font-size: 10px }
p { line-height: 120% }  /* 120% of 'font-size' */
```

4.3.4 URL + URN = URI

URLs (Uniform Resource Locators, see [RFC1738] and [RFC1808]) provide the address of a resource on the Web. Another way of identifying resources is called URN (Uniform Resource Name). Together they are called URIs (Uniform Resource Identifiers, see [URI]). This specification uses the term URI.

URI values in this specification are denoted by <uri>. The functional notation used to designate URIs in property values is "url()", as in:

Example(s):

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

The format of a URI value is 'url(' followed by optional whitespace [p. 40] followed by an optional single quote (') or double quote (") character followed by the URI itself, followed by an optional single quote (') or double quote (") character followed by optional whitespace followed by ')'. The two quote characters must be the same.

Example(s):

An example without quotes:

```
li { list-style: url(http://www.example.com/redball.png) disc }
```

Parentheses, commas, whitespace characters, single quotes (') and double quotes (") appearing in a URI must be escaped with a backslash: '\(', '\)', '\,'.

Depending on the type of URI, it might also be possible to write the above characters as URI-escapes (where "(" = %28, ")" = %29, etc.) as described in [URI].

In order to create modular style sheets that are not dependent on the absolute location of a resource, authors may use relative URIs. Relative URIs (as defined in [RFC1808]) are resolved to full URIs using a base URI. RFC 1808, section 3, defines the normative algorithm for this process. For CSS style sheets, the base URI is that of the style sheet, not that of the source document.

Example(s):

For example, suppose the following rule:

```
body { background: url("yellow") }
```

is located in a style sheet designated by the URI:

```
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 URI

```
http://www.example.org/style/yellow
```

User agents may vary in how they handle URIs that designate unavailable or inapplicable resources.

4.3.5 Counters

Counters are denoted by identifiers (see the 'counter-increment' and 'counter-reset' properties). To refer to the value of a counter, the notation 'counter(<identifier>)' or 'counter(<identifier>, <list-style-type>)' is used. The default style is 'decimal'.

To refer to a sequence of nested counters of the same name, the notation is 'counters(<identifier>, <string>)' or 'counters(<identifier>, <string>, <list-style-type>)'. See "Nested counters and scope" [p. 186] in the chapter on generated content [p. 177] .

In CSS2, the values of counters can only be referred to from the 'content' property. Note that 'none' is a possible <list-style-type>: 'counter(x, none)' yields an empty string.

Example(s):

Here is a style sheet that numbers paragraphs (P) for each chapter (H1). The paragraphs are numbered with roman numerals, followed by a period and a space:

```
P {counter-increment: par-num}
H1 {counter-reset: par-num}
P:before {content: counter(par-num, upper-roman) ". "}
```

Counters that are not in the scope [p. 186] of any 'counter-reset', are assumed to have been reset to 0 by a 'counter-reset' on the root element.

4.3.6 Colors

A <color> is either a keyword or a numerical RGB specification.

The list of keyword color names is: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, orange, purple, red, silver, teal, white, and yellow. These 17 colors have the following values:

maroon #800000red #ff0000orange #ffA500yellow #ffff00olive #808000
purple #800080 fuchsia #ff00ff white #ffffff lime #00ff00 green #008000
navy #000080 blue #0000ff aqua #00ffff teal #008080
black #000000 silver #c0c0c0 gray #808080

In addition to these color keywords, users may specify keywords that correspond to the colors used by certain objects in the user's environment. Please consult the section on system colors [p. 260] for more information.

Example(s):

```
body {color: black; background: white }
h1 { color: maroon }
h2 { color: olive }
```

The RGB color model is used in numerical color specifications. These examples all specify the same color:

Example(s):

```
em { color: #f00 }              /* #rgb */
em { color: #ff0000 }           /* #rrggbb */
em { color: rgb(255,0,0) }
em { color: rgb(100%, 0%, 0%) }
```


The format of an RGB value in hexadecimal notation is a '#' immediately followed by either three or six hexadecimal characters. The three-digit RGB notation (#rgb) is converted into six-digit form (#rrggbb) by replicating digits, not by adding zeros. For example, #fb0 expands to #ffbb00. This ensures that white (#ffffff) can be specified with the short notation (#fff) and removes any dependencies on the color depth of the display.

The format of an RGB value in the functional notation is 'rgb(' followed by a comma-separated list of three numerical values (either three integer values or three percentage values) followed by ')'. The integer value 255 corresponds to 100%, and to F or FF in the hexadecimal notation: rgb(255,255,255) = rgb(100%,100%,100%) = #FFF. Whitespace [p. 40] characters are allowed around the numerical values.

All RGB colors are specified in the sRGB color space (see [SRGB]). User agents may vary in the fidelity with which they represent these colors, but using sRGB provides an unambiguous and objectively measurable definition of what the color should be, which can be related to international standards (see [COLORIMETRY]).

Conforming user agents may limit their color-displaying efforts to performing a gamma-correction on them. sRGB specifies a display gamma of 2.2 under specified viewing conditions. User agents should adjust the colors given in CSS such that, in combination with an output device's "natural" display gamma, an effective display gamma of 2.2 is produced. See the section on gamma correction [p. 212] for further details. Note that only colors specified in CSS are affected; e.g., images are expected to carry their own color information.

Values outside the device gamut should be clipped: the red, green, and blue values must be changed to fall within the range supported by the device. For a typical CRT monitor, whose device gamut is the same as sRGB, the four rules below are equivalent:

Example(s):

```
em { color: rgb(255,0,0) }       /* integer range 0 - 255 */
em { color: rgb(300,0,0) }       /* clipped to rgb(255,0,0) */
em { color: rgb(255,-10,0) }     /* clipped to rgb(255,0,0) */
em { color: rgb(110%, 0%, 0%) }  /* clipped to rgb(100%,0%,0%) */
```

Other devices, such as printers, have different gamuts than sRGB; some colors outside the 0..255 sRGB range will be representable (inside the device gamut), while other colors inside the 0..255 sRGB range will be outside the device gamut and will thus be clipped.

4.3.7 Strings

Strings can either be written with double quotes or with single quotes. Double quotes cannot occur inside double quotes, unless escaped (as '\"' or as '\22'). Analogously for single quotes ("\'" or "\27").

Example(s):

```
"this is a 'string'"
"this is a \"string\""
'this is a "string"'
'this is a \'string\''
```

A string cannot directly contain a newline. To include a newline in a string, use the escape "\A" (hexadecimal A is the line feed character in Unicode, but represents the generic notion of "newline" in CSS). See the 'content' property for an example.

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

Example(s):

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

4.3.8 Unsupported Values

If a UA does not support a particular value, it should *ignore* that value when parsing stylesheets, as if that value was an illegal value [p. 47] . For example:

Example(s):

```
h3 {
  display: inline;
  display: run-in;
}
```

A UA that supports the 'run-in' value for the 'display' property will accept the first display declaration and then "write over" that value with the second display declaration. A UA that does not support the 'run-in' value will process the first display declaration and ignore the second display declaration.

4.4 CSS document representation

A CSS style sheet is a sequence of characters from the Universal Character Set (see [ISO10646]). For transmission and storage, these characters must be encoded by a character encoding that supports the set of characters available in US-ASCII (e.g., ISO 8859-x, SHIFT JIS, etc.). For a good introduction to character sets and character encodings, please consult the HTML 4.0 specification ([HTML40], chapter 5), See also the XML 1.0 specification ([XML10], sections 2.2 and 4.3.3, and Appendix F.

When a style sheet is embedded in another document, such as in the STYLE element or "style" attribute of HTML, the style sheet shares the character encoding of the whole document.

When a style sheet resides in a separate file, user agents must observe the following priorities when determining a document's character encoding (from highest priority to lowest):

1. An HTTP "charset" parameter in a "Content-Type" field.
2. The @charset at-rule.
3. Mechanisms of the language of the referencing document (e.g., in HTML, the "charset" attribute of the LINK element).

At most one @charset rule may appear in an external style sheet — it must *not* appear in an embedded style sheet — and it must appear at the very start of the document, not preceded by any characters. After "@charset", authors specify the name of a character encoding. The name must be a charset name as described in the IANA registry (See [IANA]. Also, see [CHARSETS] for a complete list of charsets). For example:

Example(s):

@charset "ISO-8859-1";

This specification does not mandate which character encodings a user agent must support.

Note that reliance on the @charset construct theoretically poses a problem since there is no *a priori* information on how it is encoded. In practice, however, the encodings in wide use on the Internet are either based on ASCII, UTF-16, UCS-4, or (rarely) on EBCDIC. This means that in general, the initial byte values of a document enable a user agent to detect the encoding family reliably, which provides enough information to decode the @charset rule, which in turn determines the exact character encoding.

4.4.1 Referring to characters not represented in a character encoding

A style sheet may have to refer to characters that cannot be represented in the current character encoding. These characters must be written as escaped [p. 42] references to ISO 10646 characters. These escapes serve the same purpose as numeric character references in HTML or XML documents (see [HTML40], chapters 5 and 25).

The character escape mechanism should be used when only a few characters must be represented this way. If most of a document requires escaping, authors should encode it with a more appropriate encoding (e.g., if the document contains a lot of Greek characters, authors might use "ISO-8859-7" or "UTF-8").

Intermediate processors using a different character encoding may translate these escaped sequences into byte sequences of that encoding. Intermediate processors must not, on the other hand, alter escape sequences that cancel the special meaning of an ASCII character.

Conforming user agents [p. 34] must correctly map to Unicode all characters in any character encodings that they recognize (or they must behave as if they did).

For example, a document transmitted as ISO-8859-1 (Latin-1) cannot contain Greek letters directly: "κουρος" (Greek: "kouros") has to be written as "\3BA\3BF\3C5\3C1\3BF\3C2".

Note. *In HTML 4.0, numeric character references are interpreted in "style" attribute values but not in the content of the STYLE element. Because of this asymmetry, we recommend that authors use the CSS character escape mechanism rather than numeric character references for both the "style" attribute and the STYLE element. For example, we recommend:*

```
<SPAN style="font-family: L\FC beck">...</SPAN>
```

rather than:

```
<SPAN style="font-family: L&#252;beck">...</SPAN>
```

Syntax and basic data types

5 Selectors

Contents

5.1 Pattern matching

In CSS, pattern matching rules determine which style rules apply to elements in the document tree [p. 33] . These patterns, called selectors, may range from simple element names to rich contextual patterns. If all conditions in the pattern are true for a certain element, the selector *matches* the element.

The case-sensitivity of document language element names in selectors depends on the document language. For example, in HTML, element names are case-insensitive, but in XML they are case-sensitive.

The following table summarizes CSS 2.1 selector syntax:

Pattern	Meaning	Described in section
*	Matches any element.	Universal selector [p. 62]
E	Matches any E element (i.e., an element of type E).	Type selectors [p. 62]
E F	Matches any F element that is a descendant of an E element.	Descendant selectors [p. 62]
E > F	Matches any F element that is a child of an element E.	Child selectors [p. 63]
E:first-child	Matches element E when E is the first child of its parent.	The :first-child pseudo-class [p. 70]
E:link E:visited	Matches element E if E is the source anchor of a hyperlink of which the target is not yet visited (:link) or already visited (:visited).	The link pseudo-classes [p. 70]
E:active E:hover E:focus	Matches E during certain user actions.	The dynamic pseudo-classes [p. 71]
E:lang(c)	Matches element of type E if it is in (human) language c (the document language specifies how language is determined).	The :lang() pseudo-class [p. 72]
E + F	Matches any F element immediately preceded by an element E.	Adjacent selectors [p. 63]
E[foo]	Matches any E element with the "foo" attribute set (whatever the value).	Attribute selectors [p. 64]
E[foo="warning"]	Matches any E element whose "foo" attribute value is exactly equal to "warning".	Attribute selectors [p. 64]
E[foo~="warning"]	Matches any E element whose "foo" attribute value is a list of space-separated values, one of which is exactly equal to "warning".	Attribute selectors [p. 64]
E[lang|="en"]	Matches any E element whose "lang" attribute has a hyphen-separated list of values beginning (from the left) with "en".	Attribute selectors [p. 64]

DIV.warning	*Language specific.* (In HTML, the same as DIV[class~="warning"].)	Class selectors [p. 66]
E#myid	Matches any E element with ID equal to "myid".	ID selectors [p. 67]

5.2 Selector syntax

A *simple selector* is either a type selector [p. 62] or universal selector [p. 62] followed immediately by zero or more attribute selectors [p. 64] , ID selectors [p. 67] , or pseudo-classes [p. 69] , in any order. The simple selector matches if all of its components match.

Note: the terminology used here in CSS 2.1 is different from what is used in CSS3. For example, a "simple selector" refers to a smaller part of a selector in CSS3 than in CSS 2.1. See the CSS3 Selectors module [CSS3SEL].

A *selector* is a chain of one or more simple selectors separated by combinators. *Combinators* are: whitespace, ">", and "+". Whitespace may appear between a combinator and the simple selectors around it.

The elements of the document tree that match a selector are called *subjects* of the selector. A selector consisting of a single simple selector matches any element satisfying its requirements. Prepending a simple selector and combinator to a chain imposes additional matching constraints, so the subjects of a selector are always a subset of the elements matching the last simple selector.

One pseudo-element [p. 69] may be appended to the last simple selector in a chain, in which case the style information applies to a subpart of each subject.

5.2.1 Grouping

When several selectors share the same declarations, they may be grouped into a comma-separated list.

Example(s):

In this example, we condense three rules with identical declarations into one. Thus,

```
h1 { font-family: sans-serif }
h2 { font-family: sans-serif }
h3 { font-family: sans-serif }
```

is equivalent to:

```
h1, h2, h3 { font-family: sans-serif }
```

CSS offers other "shorthand" mechanisms as well, including multiple declarations [p. 45] and shorthand properties [p. 19] .

5.3 Universal selector

The universal selector, written "*", matches the name of any element type. It matches any single element in the document tree. [p. 33]

If the universal selector is not the only component of a simple selector [p. 61] , the "*" may be omitted. For example:

- `*[lang=fr]` and `[lang=fr]` are equivalent.
- `*.warning` and `.warning` are equivalent.
- `*#myid` and `#myid` are equivalent.

5.4 Type selectors

A *type selector* matches the name of a document language element type. A type selector matches every instance of the element type in the document tree.

Example(s):

The following rule matches all H1 elements in the document tree:

```
h1 { font-family: sans-serif }
```

5.5 Descendant selectors

At times, authors may want selectors to match an element that is the descendant of another element in the document tree (e.g., "Match those EM elements that are contained by an H1 element"). Descendant selectors express such a relationship in a pattern. A descendant selector is made up of two or more selectors separated by whitespace [p. 40] . A descendant selector of the form "`A B`" matches when an element `B` is an arbitrary descendant of some ancestor [p. 33] element `A`.

Example(s):

For example, consider the following rules:

```
h1 { color: red }
em { color: red }
```

Although the intention of these rules is to add emphasis to text by changing its color, the effect will be lost in a case such as:

```
<H1>This headline is <EM>very</EM> important</H1>
```

We address this case by supplementing the previous rules with a rule that sets the text color to blue whenever an EM occurs anywhere within an H1:

```
h1 { color: red }
em { color: red }
h1 em { color: blue }
```

The third rule will match the EM in the following fragment:

```
<H1>This <SPAN class="myclass">headline
is <EM>very</EM> important</SPAN></H1>
```

Example(s):

The following selector:

```
div * p
```

matches a P element that is a grandchild or later descendant of a DIV element. Note the whitespace on either side of the "*" is not part of the universal selector; the whitespace is the descendant selector indicating that the DIV must be the ancestor of some element, and that that element must be an ancestor of the P.

Example(s):

The selector in the following rule, which combines descendant and attribute selectors [p. 64] , matches any element that (1) has the "href" attribute set and (2) is inside a P that is itself inside a DIV:

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

5.6 Child selectors

A *child selector* matches when an element is the child [p. 33] of some element. A child selector is made up of two or more selectors separated by ">".

Example(s):

The following rule sets the style of all P elements that are children of BODY:

```
body > P { line-height: 1.3 }
```

Example(s):

The following example combines descendant selectors and child selectors:

```
div ol>li p
```

It matches a P element that is a descendant of an LI; the LI element must be the child of an OL element; the OL element must be a descendant of a DIV. Notice that the optional whitespace around the ">" combinator has been left out.

For information on selecting the first child of an element, please see the section on the :first-child [p. 70] pseudo-class below.

5.7 Adjacent sibling selectors

Adjacent sibling selectors have the following syntax: E1 + E2, where E2 is the subject of the selector. The selector matches if E1 and E2 share the same parent in the document tree and E1 immediately precedes E2, ignoring non-element nodes (such as text nodes and comments).

In some contexts, adjacent elements generate formatting objects whose presentation is handled automatically (e.g., collapsing vertical margins between adjacent boxes). The "+" selector allows authors to specify additional style to adjacent elements.

Example(s):

Thus, the following rule states that when a P element immediately follows a MATH element, it should not be indented:

```
math + p { text-indent: 0 }
```

The next example reduces the vertical space separating an H1 and an H2 that immediately follows it:

```
h1 + h2 { margin-top: -5mm }
```

Example(s):

The following rule is similar to the one in the previous example, except that it adds a class selector. Thus, special formatting only occurs when H1 has `class="opener"`:

```
h1.opener + h2 { margin-top: -5mm }
```

5.8 Attribute selectors

CSS 2.1 allows authors to specify rules that match attributes defined in the source document.

5.8.1 Matching attributes and attribute values

Attribute selectors may match in four ways:

`[att]`
: Match when the element sets the "att" attribute, whatever the value of the attribute.

`[att=val]`
: Match when the element's "att" attribute value is exactly "val".

`[att~=val]`
: Match when the element's "att" attribute value is a space-separated list of "words", one of which is exactly "val". If this selector is used, the words in the value must not contain spaces (since they are separated by spaces).

`[att|=val]`
: Match when the element's "att" attribute value is a hyphen-separated list of "words", beginning with "val". The match always starts at the beginning of the attribute value. This is primarily intended to allow language subcode matches (e.g., the "lang" attribute in HTML) as described in RFC 1766 ([RFC1766]).

Attribute values must be identifiers or strings. The case-sensitivity of attribute names and values in selectors depends on the document language.

Example(s):

For example, the following attribute selector matches all H1 elements that specify the "title" attribute, whatever its value:

```
h1[title] { color: blue; }
```

Example(s):

In the following example, the selector matches all SPAN elements whose "class" attribute has exactly the value "example":

```
span[class=example] { color: blue; }
```

Multiple attribute selectors can be used to refer to several attributes of an element, or even several times to the same attribute.

Example(s):

Here, the selector matches all SPAN elements whose "hello" attribute has exactly the value "Cleveland" and whose "goodbye" attribute has exactly the value "Columbus":

```
span[hello="Cleveland"][goodbye="Columbus"] { color: blue; }
```

Example(s):

The following selectors illustrate the differences between "=" and "~=". The first selector will match, for example, the value "copyright copyleft copyeditor" for the "rel" attribute. The second selector will only match when the "href" attribute has the value "http://www.w3.org/".

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

Example(s):

The following rule hides all elements for which the value of the "lang" attribute is "fr" (i.e., the language is French).

```
*[lang=fr] { display : none }
```

Example(s):

The following rule will match for values of the "lang" attribute that begin with "en", including "en", "en-US", and "en-cockney":

```
*[lang|="en"] { color : red }
```

Example(s):

Similarly, the following aural style sheet rules allow a script to be read aloud in different voices for each role:

```
DIALOGUE[character=romeo]
     { voice-family: "Lawrence Olivier", charles, male }

DIALOGUE[character=juliet]
     { voice-family: "Vivien Leigh", victoria, female }
```

5.8.2 Default attribute values in DTDs

Matching takes place on attribute values in the document tree. Default attribute values may be defined in a DTD or elsewhere, but cannot be selected by attribute selectors. Style sheets should be designed so that they work even if the default values are not included in the document tree.

Example(s):

For example, consider an element EXAMPLE with an attribute "notation" that has a default value of "decimal". The DTD fragment might be

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

If the style sheet contains the rules

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

then to catch the cases where this attribute is set by default, and not explicitly, the following rule might be added:

```
EXAMPLE { /*... default property settings ...*/ }
```

Because this selector is less specific [p. 84] than an attribute selector, it will only be used for the default case. Care has to be taken that all other attribute values that don't get the same style as the default are explicitly covered.

5.8.3 Class selectors

Working with HTML, authors may use the period (`.`) notation as an alternative to the `~=` notation when representing the `class` attribute. Thus, for HTML, `div.value` and `div[class~=value]` have the same meaning. The attribute value must immediately follow the "period" (`.`). UAs may apply selectors using the period (.) notation in XML documents if the UA has namespace specific knowledge that allows it to determine which attribute is the "class" attribute for the respective namespace. One such example of namespace specific knowledge is the prose in the specification for a particular namespace (e.g. SVG 1.0 [SVG10] describes the SVG "class" attribute [p. ??] and how a UA should interpret it, and similarly MathML 2.0 [MATH20] describes the MathML "class" attribute [p. ??] .)

Example(s):

For example, we can assign style information to all elements with `class~="pastoral"` as follows:

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

or just

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

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

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

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

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

To match a subset of "class" values, each value must be preceded by a ".", in any order.

Example(s):

For example, the following rule matches any P element whose "class" attribute has been assigned a list of space-separated values that includes "pastoral" and "marine":

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

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

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

5.9 ID selectors

Document languages may contain attributes that are declared to be of type ID. What makes attributes of type ID special is that no two such attributes can have the same value; whatever the document language, an ID attribute can be used to uniquely identify its element. In HTML all ID attributes are named "id"; XML applications may name ID attributes differently, but the same restriction applies.

The ID attribute of a document language allows authors to assign an identifier to one element instance in the document tree. CSS ID selectors match an element instance based on its identifier. A CSS ID selector contains a "#" immediately followed by the ID value.

Example(s):

The following ID selector matches the H1 element whose ID attribute has the value "chapter1":

```
h1#chapter1 { text-align: center }
```

In the following example, the style rule matches the element that has the ID value "z98y". The rule will thus match for the P element:

```
<HEAD>
  <TITLE>Match P</TITLE>
  <STYLE type="text/css">
    *#z98y { letter-spacing: 0.3em }
  </STYLE>
</HEAD>
<BODY>
   <P id=z98y>Wide text</P>
</BODY>
```

In the next example, however, the style rule will only match an H1 element that has an ID value of "z98y". The rule will not match the P element in this example:

```
<HEAD>
  <TITLE>Match H1 only</TITLE>
  <STYLE type="text/css">
    H1#z98y { letter-spacing: 0.5em }
  </STYLE>
</HEAD>
<BODY>
   <P id=z98y>Wide text</P>
</BODY>
```

ID selectors have a higher specificity than attribute selectors. For example, in HTML, the selector `#p123` is more specific than `[id=p123]` in terms of the cascade [p. 79] .

Note. *In XML 1.0 [XML10], the information about which attribute contains an element's IDs is contained in a DTD. When parsing XML, UAs do not always read the DTD, and thus may not know what the ID of an element is. If a style sheet designer knows or suspects that this will be the case, he should use normal attribute selectors instead:* `[name=p371]` *instead of* `#p371`*. However, the cascading order of normal attribute selectors is different from ID selectors. It may be necessary to add an "!important" priority to the declarations:* `[name=p371] {color: red ! important}`*. Of course, elements in XML 1.0 documents without a DTD do not have IDs at all.*

5.10 Pseudo-elements and pseudo-classes

In CSS 2.1, style is normally attached to an element based on its position in the document tree [p. 33] . This simple model is sufficient for many cases, but some common publishing scenarios may not be possible due to the structure of the document tree [p. 33] . For instance, in HTML 4.0 (see [HTML40]), no element refers to the first line of a paragraph, and therefore no simple CSS selector may refer to it.

CSS introduces the concepts of *pseudo-elements* and *pseudo-classes* to permit formatting based on information that lies outside the document tree.

- Pseudo-elements create abstractions about the document tree beyond those specified by the document language. For instance, document languages do not offer mechanisms to access the first letter or first line of an element's content. CSS pseudo-elements allow style sheet designers to refer to this otherwise inaccessible information. Pseudo-elements may also provide style sheet designers a way to assign style to content that does not exist in the source document (e.g., the :before and :after [p. 177] pseudo-elements give access to generated content).
- Pseudo-classes classify elements on characteristics other than their name, attributes or content; in principle characteristics that cannot be deduced from the document tree. Pseudo-classes may be dynamic, in the sense that an element may acquire or lose a pseudo-class while a user interacts with the document. The exceptions are ':first-child' [p. 70] , which *can* be deduced from the document tree, and ':lang()' [p. 72] , which can be deduced from the document tree in some cases.

Neither pseudo-elements nor pseudo-classes appear in the document source or document tree.

Pseudo-classes are allowed anywhere in selectors while pseudo-elements may only appear after the subject [p. 61] of the selector.

Pseudo-element and pseudo-class names are case-insensitive.

Some pseudo-classes are mutually exclusive, while others can be applied simultaneously to the same element. In case of conflicting rules, the normal cascading order [p. 83] determines the outcome.

Conforming HTML user agents [p. 34] may ignore [p. 46] all rules with :first-line or :first-letter in the selector, or, alternatively, may only support a subset of the properties on these pseudo-elements.

5.11 Pseudo-classes

5.11.1 :first-child pseudo-class

The :first-child pseudo-class matches an element that is the first child of some other element.

Example(s):

In the following example, the selector matches any P element that is the first child of a DIV element. The rule suppresses indentation for the first paragraph of a DIV:

```
div > p:first-child { text-indent: 0 }
```

This selector would match the P inside the DIV of the following fragment:

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

but would not match the second P in the following fragment:

```
<P> The last P before the note.
<DIV class="note">
   <H2>Note</H2>
   <P> The first P inside the note.
</DIV>
```

Example(s):

The following rule sets the font weight to 'bold' for any EM element that is some descendant of a P element that is a first child:

```
p:first-child em { font-weight : bold }
```

Note that since anonymous [p. 111] boxes are not part of the document tree, they are not counted when calculating the first child.

For example, the EM in:

```
<P>abc <EM>default</EM>
```

is the first child of the P.

The following two selectors are equivalent:

```
* > a:first-child   /* A is first child of any element */
a:first-child       /* Same */
```

5.11.2 The link pseudo-classes: :link and :visited

User agents commonly display unvisited links differently from previously visited ones. CSS provides the pseudo-classes ':link' and ':visited' to distinguish them:

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

Note. *After a certain amount of time, user agents may choose to return a visited link to the (unvisited) ':link' state.*

The two states are mutually exclusive.

The document language determines which elements are hyperlink source anchors. For example, in HTML 4.0, the link pseudo-classes apply to A elements with an "href" attribute. Thus, the following two CSS 2.1 declarations have similar effect:

```
a:link { color: red }
:link  { color: red }
```

Example(s):

If the following link:

```
<A class="external" href="http://out.side/">external link</A>
```

has been visited, this rule:

```
a.external:visited { color: blue }
```

will cause it to be blue.

Note. It is possible for stylesheet authors to abuse the :link and :visited pseudo-classes to determine which sites a user has visited without the user's consent. UAs may therefore treat all links as unvisited links, or implement other measures to preserve the user's privacy while rendering visited and unvisited links differently. See [P3P] for more information about handling privacy.

5.11.3 The dynamic pseudo-classes: :hover, :active, and :focus

Interactive user agents sometimes change the rendering in response to user actions. CSS provides three pseudo-classes for common cases:

- The :hover pseudo-class applies while the user designates an element (with some pointing device), but does not activate it. For example, a visual user agent could apply this pseudo-class when the cursor (mouse pointer) hovers over a box generated by the element. User agents not supporting interactive media [p. 89] do not have to support this pseudo-class. Some conforming user agents supporting interactive media [p. 89] may not be able to support this pseudo-class (e.g., a pen device).
- The :active pseudo-class applies while an element is being activated by the user. For example, between the times the user presses the mouse button and releases it.
- The :focus pseudo-class applies while an element has the focus (accepts keyboard events or other forms of text input).

An element may match several pseudo-classes at the same time.

CSS doesn't define which elements may be in the above states, or how the states are entered and left. Scripting may change whether elements react to user events or not, and different devices and UAs may have different ways of pointing to, or activating elements.

User agents are not required to reflow a currently displayed document due to pseudo-class transitions. For instance, a style sheet may specify that the 'font-size' of an :active link should be larger than that of an inactive link, but since this may cause letters to change position when the reader selects the link, a UA may ignore the corresponding style rule.

Example(s):

```
a:link    { color: red }    /* unvisited links */
a:visited { color: blue }   /* visited links   */
a:hover   { color: yellow } /* user hovers     */
a:active  { color: lime }   /* active links    */
```

Note that the A:hover must be placed after the A:link and A:visited rules, since otherwise the cascading rules will hide the 'color' property of the A:hover rule. Similarly, because A:active is placed after A:hover, the active color (lime) will apply when the user both activates and hovers over the A element.

Example(s):

An example of combining dynamic pseudo-classes:

```
a:focus { background: yellow }
a:focus:hover { background: white }
```

The last selector matches A elements that are in pseudo-class :focus and in pseudo-class :hover.

For information about the presentation of focus outlines, please consult the section on dynamic focus outlines [p. 262] .

Note. *In CSS1, the ':active' pseudo-class was mutually exclusive with ':link' and ':visited'. That is no longer the case. An element can be both ':visited' and ':active' (or ':link' and ':active') and the normal cascading rules determine which properties apply.*

5.11.4 The language pseudo-class: :lang

If the document language specifies how the human language of an element is determined, it is possible to write selectors in CSS that match an element based on its language. For example, in HTML [HTML40], the language is determined by a combination of the "lang" attribute, the META element, and possibly by information from the protocol (such as HTTP headers). XML uses an attribute called xml:lang, and there may be other document language-specific methods for determining the language.

The pseudo-class ':lang(C)' matches if the element is in language C. Here C is a language code as specified in HTML 4.0 [HTML40] and RFC 1766 [RFC1766]. It is matched the same way as for the '|=' operator [p. 64] .

Example(s):

The following rules set the quotation marks for an HTML document that is either in French or German:

```
html:lang(fr) { quotes: '« ' ' »' }
html:lang(de) { quotes: '»' '«' '\2039' '\203A' }
:lang(fr) > Q { quotes: '« ' ' »' }
:lang(de) > Q { quotes: '»' '«' '\2039' '\203A' }
```

The second pair of rules actually set the 'quotes' property on Q elements according to the language of its parent. This is done because the choice of quote marks is typically based on the language of the element around the quote, not the quote itself: like this piece of French "à l'improviste" in the middle of an English text uses the English quotation marks.

5.12 Pseudo-elements

5.12.1 The :first-line pseudo-element

The :first-line pseudo-element applies special styles to the contents of the first formatted line of a paragraph. For instance:

```
p:first-line { text-transform: uppercase }
```

The above rule means "change the letters of the first line of every paragraph to uppercase". However, the selector "P:first-line" does not match any real HTML element. It does match a pseudo-element that conforming user agents [p. 34] will insert at the beginning of every paragraph.

Note that the length of the first line depends on a number of factors, including the width of the page, the font size, etc. Thus, an ordinary HTML paragraph such as:

```
<P>This is a somewhat long HTML
paragraph that will be broken into several
lines. The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

the lines of which happen to be broken as follows:

```
THIS IS A SOMEWHAT LONG HTML PARAGRAPH THAT
will be broken into several lines. The first
line will be identified by a fictional tag
sequence. The other lines will be treated as
ordinary lines in the paragraph.
```


might be "rewritten" by user agents to include the *fictional tag sequence* for :first-line. This fictional tag sequence helps to show how properties are inherited.

```
<P><P:first-line> This is a somewhat long HTML
paragraph that </P:first-line> will be broken into several
lines. The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

If a pseudo-element breaks up a real element, the desired effect can often be described by a fictional tag sequence that closes and then re-opens the element. Thus, if we mark up the previous paragraph with a SPAN element:

```
<P><SPAN class="test"> This is a somewhat long HTML
paragraph that will be broken into several
lines.</SPAN> The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

the user agent could generate the appropriate start and end tags for SPAN when inserting the fictional tag sequence for :first-line.

```
<P><P:first-line><SPAN class="test"> This is a
somewhat long HTML
paragraph that will </SPAN></P:first-line><SPAN class="test"> be
broken into several
lines.</SPAN> The first line will be identified
by a fictional tag sequence. The other lines
will be treated as ordinary lines in the
paragraph.</P>
```

The :first-line pseudo-element can only be attached to a block-level element.

The "first formatted line" of an element may occur inside a block-level descendant in the same flow (i.e., a block-level descendant that is not positioned and not a float). E.g., the first line of the DIV in `<DIV><P>This line...</P></DIV>` is the first line of the P (assuming that both P and DIV are block-level).

A UA should act as if the fictional start tag of the first-line pseudo-element is just inside the smallest enclosing block-level element. (Since CSS1 and CSS2 were silent on this case, authors should not rely on this behavior.) Here is an example. The fictional tag sequence for

```
<DIV>
  <P>First paragraph</P>
  <P>Second paragraph</P>
</DIV>
```

is

```
<DIV>
  <P><DIV:first-line><P:first-line>First paragraph</P:first-line></DIV:first-line></P>
  <P><P:first-line>Second paragraph</P:first-line></P>
</DIV>
```

The :first-line pseudo-element is similar to an inline-level element, but with certain restrictions. Only the following properties apply to a :first-line pseudo-element: font properties, [p. 213] color properties, [p. 205] background properties, [p. 206] 'word-spacing', 'letter-spacing', 'text-decoration', 'vertical-align', 'text-transform', 'line-height',.

5.12.2 The :first-letter pseudo-element

The :first-letter pseudo-element may be used for "initial caps" and "drop caps", which are common typographical effects. This type of initial letter is similar to an inline-level element if its 'float' property is 'none', otherwise it is similar to a floated element.

These are the properties that apply to :first-letter pseudo-elements: font properties, [p. 213] 'text-decoration', 'text-transform', 'letter-spacing', 'word-spacing' (when appropriate), 'line-height', 'vertical-align' (only if 'float' is 'none'), margin properties, [p. 95] padding properties, [p. 98] border properties, [p. 100] color properties, [p. 205] background properties. [p. 206] To allow UAs to render a typographically correct drop cap or initial cap, the UA may choose a line-height, width and height based on the shape of the letter, unlike for normal elements. CSS3 is expected to have specific properties that apply to first-letter.

This example shows a possible rendering of an initial cap. Note that the 'line-height' that is inherited by the first-letter pseudo-element is 1.1, but the UA in this example has computed the height of the first letter differently, so that it doesn't cause any unnecessary space between the first two lines. Also note that the fictional start tag of the first letter is inside the SPAN, and thus the font weight of the first letter is normal, not bold as the SPAN:

```
p { line-height: 1.1 }
p:first-letter { font-size: 3em; font-weight: normal }
span { font-weight: bold }
...
<p><span>Het hemelsche</span> gerecht heeft zich ten lange lesten<br>
Erbarremt over my en mijn benaeuwde vesten<br>
En arme burgery, en op mijn volcx gebed<br>
En dagelix geschrey de bange stad ontzet.
```

Het hemelsche gerecht heeft zich ten lange lesten
Erbarremt over my en mijn benaeuwde vesten
En arme burgery, en op mijn volcx gebed
En dagelix geschrey de bange stad ontzet.

The following CSS 2.1 will make a drop cap initial letter span about two lines:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
 <HEAD>
  <TITLE>Drop cap initial letter</TITLE>
  <STYLE type="text/css">
   P              { font-size: 12pt; line-height: 1.2 }
   P:first-letter { font-size: 200%; font-style: italic;
                    font-weight: bold; float: left }
   SPAN           { text-transform: uppercase }
  </STYLE>
 </HEAD>
 <BODY>
  <P><SPAN>The first</SPAN> few words of an article
    in The Economist.</P>
 </BODY>
</HTML>
```

This example might be formatted as follows:

THE FIRST few
words of an
article in the
Economist

The fictional tag sequence is:

```
<P>
<SPAN>
<P:first-letter>
T
</P:first-letter>he first
</SPAN>
few words of an article in the Economist.
</P>
```

Note that the :first-letter pseudo-element tags abut the content (i.e., the initial character), while the :first-line pseudo-element start tag is inserted right after the start tag of the element to which it is attached.

In order to achieve traditional drop caps formatting, user agents may approximate font sizes, for example to align baselines. Also, the glyph outline may be taken into account when formatting.

Punctuation (i.e, characters defined in Unicode [UNICODE] in the "open" (Ps), "close" (Pe), and "other" (Po) punctuation classes), that precedes the first letter should be included, as in:

"A bird in
the hand
is worth
two in the bush,"
says an old proverb.

The :first-letter pseudo-element can be used with all elements that contain text, or that have a descendant in the same flow that contains text. A UA should act as if the fictional start tag of the first-letter pseudo-element is just before the first text of the element, even if that first text is in a descendant.

Example(s):

Here is an example. The fictional tag sequence for this HTML fragment:

```
<div>
<p>The first text.
```

is:

```
<div>
<p><div:first-letter><p:first-letter>T</.../></...>he first text.
```

Some languages may have specific rules about how to treat certain letter combinations. In Dutch, for example, if the letter combination "ij" appears at the beginning of a word, both letters should be considered within the :first-letter pseudo-element.

If the letters that would form the first-letter are not in the same element, such as "'T" in `<p>'<em>T...`, the UA may create a first-letter pseudo-element from one of the elements, both elements, or simply not create a pseudo-element.

Example(s):

The following example illustrates how overlapping pseudo-elements may interact. The first letter of each P element will be green with a font size of '24pt'. The rest of the first formatted line will be 'blue' while the rest of the paragraph will be 'red'.

```
p { color: red; font-size: 12pt }
p:first-letter { color: green; font-size: 200% }
p:first-line { color: blue }

<P>Some text that ends up on two lines</P>
```

Assuming that a line break will occur before the word "ends", the fictional tag sequence for this fragment might be:

```
<P>
<P:first-line>
<P:first-letter>
S
</P:first-letter>ome text that
</P:first-line>
ends up on two lines
</P>
```

Note that the :first-letter element is inside the :first-line element. Properties set on :first-line are inherited by :first-letter, but are overridden if the same property is set on :first-letter.

5.12.3 The :before and :after pseudo-elements

The ':before' and ':after' pseudo-elements can be used to insert generated content before or after an element's content. They are explained in the section on generated text. [p. 177]

Example(s):

```
h1:before {content: counter(chapno, upper-roman) ". "}
```

When the :first-letter and :first-line pseudo-elements are combined with :before and :after, they apply to the first letter or line of the element including the inserted text.

Example(s):

```
p.special:before {content: "Special! "}
p.special:first-letter {color: #ffd800}
```

This will render the "S" of "Special!" in gold.

---

6 Assigning property values, Cascading, and Inheritance

Contents

6.1 Specified, computed, and actual values

Once a user agent has parsed a document and constructed a document tree [p. 33] , it must assign, for every element in the tree, a value to every property that applies to the target media type [p. 87] .

The final value of a property is the result of a three-step calculation: the value is determined through specification (the "specified value"), then resolved into an absolute value if necessary (the "computed value"), and finally transformed according to the limitations of the local environment (the "actual value").

6.1.1 Specified values

User agents must first assign a specified value to a property based on the following mechanisms (in order of precedence):

1. If the cascade [p. 82] results in a value, use it.
2. Otherwise, if the property is inherited [p. 80] and the element is not the root of the document tree, use the computed value of the parent element.
3. Otherwise use the property's initial value. The initial value of each property is indicated in the property's definition.

Since it has no parent, the root of the document tree [p. 33] cannot use values from the parent element; in this case, the initial value is used if necessary.

6.1.2 Computed values

Specified values may be absolute (i.e., they are not specified relative to another value, as in 'red' or '2mm') or relative (i.e., they are specified relative to another value, as in 'auto', '2em', and '12%'). For absolute values, no computation is needed to find the computed value.

Relative values, on the other hand, must be transformed into computed values: percentages must be multiplied by a reference value (each property defines which value that is), values with relative units (em, ex, px) must be made absolute by multiplying with the appropriate font or pixel size, 'auto' values must be computed by the formulas given with each property, certain keywords ('smaller', 'bolder', 'inherit') must be replaced according to their definitions.

When the specified value is not 'inherit', the computed value of a property is determined as specified by the Computed Value line in the definition of the property. See the section on inheritance [p. 80] for the definition of computed values when the specified value is 'inherit'.

The computed value exists even when the property doesn't apply, as defined by the 'Applies To' [add reference] line. However, some properties may define the computed value of a property for an element to depend on whether the property applies to that element.

6.1.3 Actual values

A computed value is in principle ready to be used, but a user agent may not be able to make use of the value in a given environment. For example, a user agent may only be able to render borders with integer pixel widths and may therefore have to approximate the computed width. The actual value is the computed value after any approximations have been applied.

6.2 Inheritance

Some values are inherited by the children of an element in the document tree [p. 33] ,as described above [p. 79] . Each property defines [p. 17] whether it is inherited or not.

Suppose there is an H1 element with an emphasizing element (EM) inside:

```
<H1>The headline <EM>is</EM> important!</H1>
```

If no color has been assigned to the EM element, the emphasized "is" will inherit the color of the parent element, so if H1 has the color blue, the EM element will likewise be in blue.

When inheritance occurs, elements inherit computed values. The computed value from the parent element becomes both the specified value and the computed value on the child.

Example(s):

For example, given the following style sheet:

```
body { font-size: 10pt }
h1 { font-size: 120% }
```

and this document fragment:

```
<BODY>
  <H1>A <EM>large</EM> heading</H1>
</BODY>
```

the 'font-size' property for the H1 element will have the computed value '12pt' (120% times 10pt, the parent's value). Since the computed value of 'font-size' is inherited, the EM element will have the computed value '12pt' as well. If the user agent does not have the 12pt font available, the actual value of 'font-size' for both H1 and EM might be, for example, '11pt'.

6.2.1 The 'inherit' value

Each property may also have a specified value of 'inherit', which means that, for a given element, the property takes the same computed value as the property for the element's parent. The 'inherit' value can be used to strengthen inherited values, and it can also be used on properties that are not normally inherited.

Example(s):

In the example below, the 'color' and 'background' properties are set on the BODY element. On all other elements, the 'color' value will be inherited and the background will be transparent. If these rules are part of the user's style sheet, black text on a white background will be enforced throughout the document.

```
body {
  color: black !important;
  background: white !important;
}

* {
  color: inherit !important;
  background: transparent !important;
}
```

6.3 The @import rule

The *'@import'* rule allows users to import style rules from other style sheets. Any @import rules must precede all rule sets in a style sheet. The '@import' keyword must be followed by the URI of the style sheet to include. A string is also allowed; it will be interpreted as if it had url(...) around it.

Example(s):

The following lines are equivalent in meaning and illustrate both '@import' syntaxes (one with "url()" and one with a bare string):

```
@import "mystyle.css";
@import url("mystyle.css");
```

So that user agents can avoid retrieving resources for unsupported media types [p. 87] , authors may specify media-dependent @import rules. These conditional imports specify comma-separated media types after the URI.

Example(s):

The following rules illustrate how @import rules can be made media-dependent:

```
@import url("fineprint.css") print;
@import url("bluish.css") projection, tv;
```

In the absence of any media types, the import is unconditional. Specifying 'all' for the medium has the same effect.

6.4 The cascade

Style sheets may have three different origins: author, user, and user agent.

- **Author**. The author specifies style sheets for a source document according to the conventions of the document language. For instance, in HTML, style sheets may be included in the document or linked externally.
- **User**: The user may be able to specify style information for a particular document. For example, the user may specify a file that contains a style sheet or the user agent may provide an interface that generates a user style sheet (or behaves as if it did).
- **User agent**: Conforming user agents [p. 34] must apply a *default style sheet* (or behave as if they did) prior to all other style sheets for a document. A user agent's default style sheet should present the elements of the document language in ways that satisfy general presentation expectations for the document language (e.g., for visual browsers, the EM element in HTML is presented using an italic font). See A sample style sheet for HTML [p. 293] for a recommended default style sheet for HTML documents.

  Note that the default style sheet may change if system settings are modified by the user (e.g., system colors). However, due to limitations in a user agent's internal implementation, it may be impossible to change the values in the default style sheet.

Style sheets from these three origins will overlap in scope, and they interact according to the cascade.

The CSS cascade assigns a weight to each style rule. When several rules apply, the one with the greatest weight takes precedence.

By default, rules in author style sheets have more weight than rules in user style sheets. Precedence is reversed, however, for "!important" rules. All user and author rules have more weight than rules in the UA's default style sheet.

Rules specified in a given style sheet override rules of the same weight imported from other style sheets. Imported style sheets can themselves import and override other style sheets, recursively, and the same precedence rules apply.

6.4.1 Cascading order

To find the value for an element/property combination, user agents must apply the following sorting order:

1. Find all declarations that apply to the element and property in question, for the target media type [p. 87] . Declarations apply if the associated selector matches [p. 59] the element in question.
2. Sort by weight (normal or important) and origin (author, user, or user agent). In ascending order:
    1. user agent style sheets
    2. user normal style sheets
    3. author normal style sheets
    4. author important style sheets
    5. user important style sheets
3. Sort by specificity [p. 84] of selector: more specific selectors will override more general ones. Pseudo-elements and pseudo-classes are counted as normal elements and classes, respectively.
4. Finally, sort by order specified: if two rules have the same weight, origin and specificity, the latter specified wins. Rules in imported style sheets are considered to be before any rules in the style sheet itself.

Apart from the "!important" setting on individual declarations, this strategy gives author's style sheets higher weight than those of the reader. It is therefore important that the user agent give the user the ability to turn off the influence of a certain style sheet, e.g., through a pull-down menu.

6.4.2 !important rules

CSS attempts to create a balance of power between author and user style sheets. By default, rules in an author's style sheet override those in a user's style sheet (see cascade rule 3).

However, for balance, an "!important" declaration (the keywords "!" and "important" follow the declaration) takes precedence over a normal declaration. Both author and user style sheets may contain "!important" declarations, and user "!important" rules override author "!important" rules. This CSS feature improves accessibility of documents by giving users with special requirements (large fonts, color combinations, etc.) control over presentation.

Declaring a shorthand property (e.g., 'background') to be "!important" is equivalent to declaring all of its sub-properties to be "!important".

Example(s):

The first rule in the user's style sheet in the following example contains an "!important" declaration, which overrides the corresponding declaration in the author's style sheet. The second declaration will also win due to being marked "!important". However, the third rule in the user's style sheet is not "!important" and will therefore lose to the second rule in the author's style sheet (which happens to set style on a shorthand property). Also, the third author rule will lose to the second author rule since the second rule is "!important". This shows that "!important" declarations have a function also within author style sheets.

```
/* From the user's style sheet */
p { text-indent: 1em ! important }
p { font-style: italic ! important }
p { font-size: 18pt }

/* From the author's style sheet */
p { text-indent: 1.5em !important }
p { font: 12pt sans-serif !important }
p { font-size: 24pt }
```

6.4.3 Calculating a selector's specificity

A selector's specificity is calculated as follows:

- count 1 if the selector is a 'style' attribute rather than a selector, 0 otherwise (= a) (In HTML, values of an element's "style" attribute are style sheet rules. These rules have no selectors, so a=1, b=0, c=0, and d=0.)
- count the number of ID attributes in the selector (= b)
- count the number of other attributes and pseudo-classes in the selector (= c)
- count the number of element names and pseudo-elements in the selector (= d)

Concatenating the four numbers a-b-c-d (in a number system with a large base) gives the specificity.

Example(s):

Some examples:

```
*             {}  /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 */
li            {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
li:first-line {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,2 */
ul li         {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul ol+li      {}  /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3 */
h1 + *[rel=up]{}  /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1 */
ul ol li.red  {}  /* a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3 */
li.red.level  {}  /* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1 */
#x34y         {}  /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
style=""          /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

```
<HEAD>
<STYLE type="text/css">
  #x97z { color: red }
</STYLE>
</HEAD>
<BODY>
<P ID=x97z style="color: green">
</BODY>
```

In the above example, the color of the P element would be green. The declaration in the "style" attribute will override the one in the STYLE element because of cascading rule 3, since it has a higher specificity.

Note: The specificity is based only on the form of the selector. In particular, a selector of the form "[id=p33]" is counted as an attribute selector (a=0, b=0, c=1, d=0), even if the id attribute is defined as an "ID" in the source document's DTD.

6.4.4 Precedence of non-CSS presentational hints

The UA may choose to honor presentational attributes in the source document. If so, these attributes are translated to the corresponding CSS rules with specificity equal to 0, and are treated as if they were inserted at the start of the author style sheet. They may therefore be overridden by subsequent style sheet rules. In a transition phase, this policy will make it easier for stylistic attributes to coexist with style sheets.

For HTML, any attribute that is not in the following list should be considered presentational: abbr, accept-charset, accept, accesskey, action, alt, archive, axis, charset, checked, cite, class, classid, code, codebase, codetype, colspan, coords, data, datetime, declare, defer, dir, disabled, enctype, for, headers, href, hreflang, http-equiv, id, ismap, label, lang, language, longdesc, maxlength, media, method, multiple, name, nohref, object, onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onload, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onreset, onselect, onsubmit, onunload, onunload, profile, prompt, readonly, rel, rev, rowspan, scheme, scope, selected, shape, span, src, standby, start, style, summary, title, type, usemap, value, valuetype, version.

For XHTML and other languages written in XML, no attribute should be considered presentational. The styling of elements and non-presentational attributes should be handled in the user agent stylesheet.

Example(s):

The following user stylesheet would override the font weight of 'b' elements in all documents, and the color of 'font' elements with color attributes in XML documents. It would not affect the color of any 'font' elements with color attributes in HTML documents:

```
b { font-weight: normal; }
font[color] { color: orange; }
```


The following, however, would override the color of font elements in all documents:

```
font[color] { color: orange ! important; }
```

7 Media types

Contents

7.1 Introduction to media types

One of the most important features of style sheets is that they specify how a document is to be presented on different media: on the screen, on paper, with a speech synthesizer, with a braille device, etc.

Certain CSS properties are only designed for certain media (e.g., the 'page-break-before' property only applies to paged media). On occasion, however, style sheets for different media types may share a property, but require different values for that property. For example, the 'font-size' property is useful both for screen and print media. The two media types are different enough to require different values for the common property; a document will typically need a larger font on a computer screen than on paper. Therefore, it is necessary to express that a style sheet, or a section of a style sheet, applies to certain media types.

7.2 Specifying media-dependent style sheets

There are currently two ways to specify media dependencies for style sheets:

- Specify the target medium from a style sheet with the @media or @import at-rules.

  Example(s):

```
@import url("fancyfonts.css") screen;
@media print {
  /* style sheet for print goes here */
}
```

- Specify the target medium within the document language. For example, in HTML 4.0 ([HTML40]), the "media" attribute on the LINK element specifies the target media of an external style sheet:


```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
   <HEAD>
      <TITLE>Link to a target medium</TITLE>
      <LINK REL="stylesheet" TYPE="text/css"
         MEDIA="print, handheld" HREF="foo.css">
   </HEAD>
   <BODY>
      <P>The body...
   </BODY>
</HTML>
```

The @import [p. 81] rule is defined in the chapter on the cascade [p. 79] .

7.2.1 The @media rule

An @media rule specifies the target media types [p. 88] (separated by commas) of a set of rules (delimited by curly braces). The @media construct allows style sheet rules for various media in the same style sheet:

```
@media print {
  body { font-size: 10pt }
}
@media screen {
  body { font-size: 13px }
}
@media screen, print {
  body { line-height: 1.2 }
}
```

7.3 Recognized media types

The names chosen for CSS media types reflect target devices for which the relevant properties make sense. The names of media types are normative. In the following list of CSS media types, the parenthetical descriptions are not normative. Likewise, the "Media" field in the description of each property is informative.

all
: Suitable for all devices.

braille
: Intended for braille tactile feedback devices.

embossed
: Intended for paged braille printers.

handheld
: Intended for handheld devices (typically small screen, limited bandwidth).

print
: Intended for paged material and for documents viewed on screen in print preview mode. Please consult the section on paged media [p. 195] for information about formatting issues that are specific to paged media.

projection
Intended for projected presentations, for example projectors. Please consult the section on paged media [p. 195] for information about formatting issues that are specific to paged media.

screen
Intended primarily for color computer screens.

speech
Intended for speech synthesizers. Note: CSS2 had a similar media type called 'aural' for this purpose. See the appendix on aural style sheets [p. 273] for details.

tty
Intended for media using a fixed-pitch character grid (such as teletypes, terminals, or portable devices with limited display capabilities). Authors should not use pixel units [p. 49] with the "tty" media type.

tv
Intended for television-type devices (low resolution, color, limited-scrollability screens, sound available).

Media type names are case-insensitive.

Media types are mutually exclusive in the sense that a user agent can only support one media type when rendering a document. However, user agents may have different modes which support different media types.

Note. *Future versions of CSS may extend this list. Authors should not rely on media type names that are not yet defined by a CSS specification.*

7.3.1 Media groups

This section is informative, not normative.

Each CSS property definition specifies the media types for which the property must be implemented by a conforming user agent [p. 34] . Since properties generally apply to several media, the "Applies to media" section of each property definition lists media groups rather than individual media types. Each property applies to all media types in the media groups listed in its definition.

CSS 2.1 defines the following media groups:

- **continuous** or **paged**.
- **visual**, **audio**, **speech**, or **tactile**.
- **grid** (for character grid devices), or **bitmap**.
- **interactive** (for devices that allow user interaction), or **static** (for those that don't).
- **all** (includes all media types)

The following table shows the relationships between media groups and media types:

Relationship between media groups and media types

Media Types	Media Groups			
	continuous/paged	**visual/audio/speech/tactile**	**grid/bitmap**	**interactive/static**
braille	continuous	tactile	grid	both
emboss	paged	tactile	grid	static
handheld	both	visual, audio, speech	both	both
print	paged	visual	bitmap	static
projection	paged	visual	bitmap	interactive
screen	continuous	visual, audio	bitmap	both
speech	continuous	speech	N/A	both
tty	continuous	visual	grid	both
tv	both	visual, audio	bitmap	both

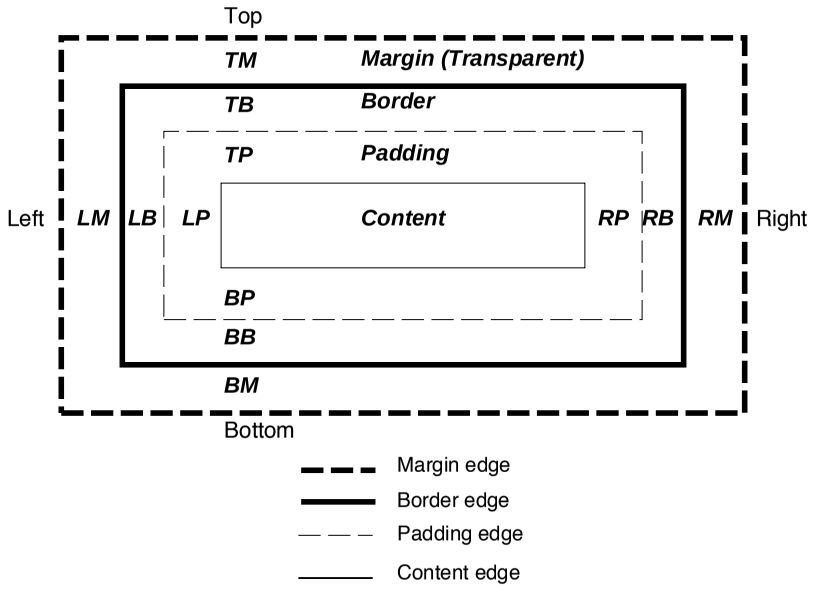
8 Box model

Contents

The CSS box model describes the rectangular boxes that are generated for elements in the document tree [p. 33] and laid out according to the visual formatting model [p. 107] .

8.1 Box dimensions

Each box has a *content area* (e.g., text, an image, etc.) and optional surrounding *padding*, *border*, and *margin* areas; the size of each area is specified by properties defined below. The following diagram shows how these areas relate and the terminology used to refer to pieces of margin, border, and padding:



The margin, border, and padding can be broken down into top, right, bottom, and left segments (e.g., in the diagram, "LM" for left margin, "RP" for right padding, "TB" for top border, etc.).

The perimeter of each of the four areas (content, padding, border, and margin) is called an "edge", so each box has four edges:

content edge or **inner edge**
: The content edge surrounds the element's rendered content [p. 32] .

padding edge
: The padding edge surrounds the box padding. If the padding has 0 width, the padding edge is the same as the content edge.

border edge
: The border edge surrounds the box's border. If the border has 0 width, the border edge is the same as the padding edge.

margin edge or **outer edge**
: The margin edge surrounds the box margin. If the margin has 0 width, the margin edge is the same as the border edge.

Each edge may be broken down into a top, right, bottom, and left edge.

The dimensions of the content area of a box — the *content width* and *content height* — depend on several factors: whether the element generating the box has the 'width' or 'height' property set, whether the box contains text or other boxes, whether

the box is a table, etc. Box widths and heights are discussed in the chapter on visual formatting model details [p. 149] .

The *box width* is given by the sum of the left and right margins, border, and padding, and the content width. The *box height* is given by the sum of the top and bottom margins, border, and padding, and the content height.

The background style of the content, padding, and border areas of a box is specified by the 'background' property of the generating element. Margin backgrounds are always transparent.

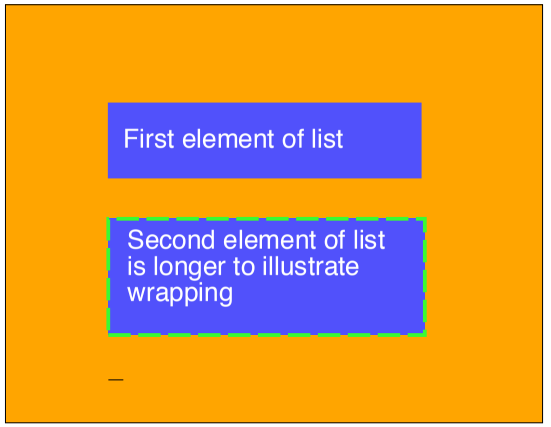
8.2 Example of margins, padding, and borders

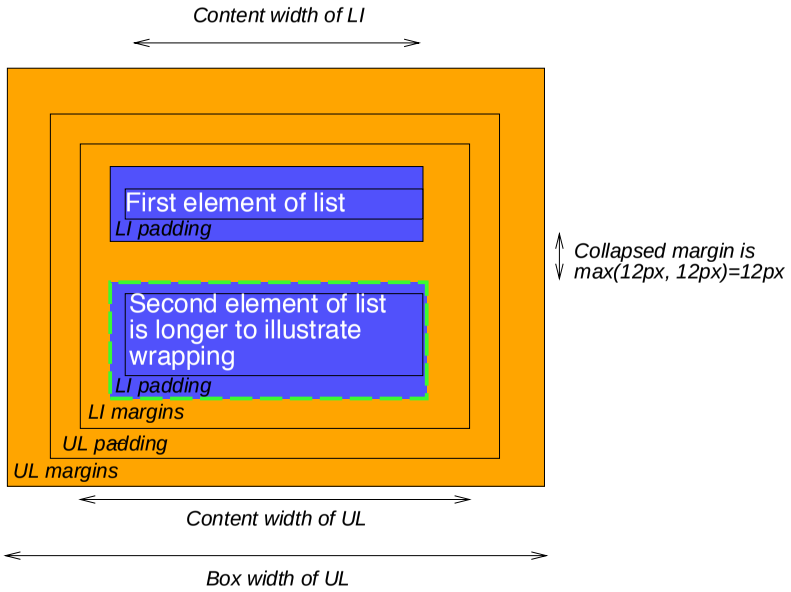
This example illustrates how margins, padding, and borders interact. The example HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Examples of margins, padding, and borders</TITLE>
    <STYLE type="text/css">
      UL {
        background: yellow;
        margin: 12px 12px 12px 12px;
        padding: 3px 3px 3px 3px;
                                    /* No borders set */
      }
      LI {
        color: white;               /* text color is white */
        background: blue;           /* Content, padding will be blue */
        margin: 12px 12px 12px 12px;
        padding: 12px 0px 12px 12px; /* Note 0px padding right */
        list-style: none            /* no glyphs before a list item */
                                    /* No borders set */
      }
      LI.withborder {
        border-style: dashed;
        border-width: medium;       /* sets border width on all sides */
        border-color: lime;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <UL>
      <LI>First element of list
      <LI class="withborder">Second element of list is longer
           to illustrate wrapping.
    </UL>
  </BODY>
</HTML>
```

results in a document tree [p. 33] with (among other relationships) a UL element that has two LI children.

The first of the following diagrams illustrates what this example would produce. The second illustrates the relationship between the margins, padding, and borders of the UL elements and those of its children LI elements.





Note that:

- The content width [p. 92] for each LI box is calculated top-down; the containing block [p. 108] for each LI box is established by the UL element.
- The height of each LI box is given by its content height [p. 92] , plus top and bottom padding, borders, and margins. Note that vertical margins between the LI boxes collapse. [p. 97]
- The right padding of the LI boxes has been set to zero width (the 'padding' property). The effect is apparent in the second illustration.
- The margins of the LI boxes are transparent — margins are always transparent — so the background color (yellow) of the UL padding and content areas shines through them.
- The second LI element specifies a dashed border (the 'border-style' property).

8.3 Margin properties: 'margin-top', 'margin-right', 'margin-bottom', 'margin-left', and 'margin'

Margin properties specify the width of the margin area [p. 91] of a box. The 'margin' shorthand property sets the margin for all four sides while the other margin properties only set their respective side. These properties apply to all elements, but vertical margins will not have any effect on non-replaced inline elements. Conforming HTML user agents [p. 34] may ignore the margin properties on the HTML element.

The properties defined in this section refer to the **<margin-width>** value type, which may take one of the following values:

<length>
Specifies a fixed width.

<percentage>
The percentage is calculated with respect to the *width* of the generated box's containing block [p. 108] . Note that this is true for 'margin-top' and 'margin-bottom' as well. If the containing block's width depends on this element, then the resulting layout is undefined in CSS 2.1.

auto
See the section on computing widths and margins [p. 153] for behavior.

Negative values for margin properties are allowed, but there may be implementation-specific limits.

'margin-top', **'margin-bottom'**

Value:	<margin-width> I inherit
Initial:	0
Applies to:	all elements but inline, non-replaced elements and internal table elements [p. 237]
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length

'margin-right', **'margin-left'**

Value:	<margin-width> I inherit
Initial:	0
Applies to:	all elements but internal table elements [p. 237]
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length

These properties set the top, right, bottom, and left margin of a box.

Example(s):

```
h1 { margin-top: 2em }
```

'margin'

Value:	<margin-width>{1,4} I inherit
Initial:	see individual properties
Applies to:	all elements but internal table elements [p. 237]
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	see individual properties

The 'margin' property is a shorthand property for setting 'margin-top', 'margin-right', 'margin-bottom', and 'margin-left' at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom margins are set to the first value and the right and left margins are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.

Example(s):

```
body { margin: 2em }         /* all margins set to 2em */
body { margin: 1em 2em }     /* top & bottom = 1em, right & left = 2em */
body { margin: 1em 2em 3em } /* top=1em, right=2em, bottom=3em, left=2em */
```

The last rule of the example above is equivalent to the example below:

```
body {
  margin-top: 1em;
  margin-right: 2em;
  margin-bottom: 3em;
  margin-left: 2em;        /* copied from opposite side (right) */
}
```

8.3.1 Collapsing margins

In this specification, the expression *collapsing margins* means that adjoining margins (no non-empty content, padding or border areas or clearance [p. 128] separate them) of two or more boxes (which may be next to one another or nested) combine to form a single margin.

In CSS 2.1, horizontal margins never collapse.

Vertical margins may collapse between certain boxes:

- Two or more adjoining vertical margins of block [p. 109] boxes in the normal flow [p. 117] collapse. The resulting margin width is the maximum of the adjoining margin widths. In the case of negative margins, the maximum of the absolute values of the negative adjoining margins is deducted from the maximum of the positive adjoining margins. If there are no positive margins, the absolute maximum of the negative adjoining margins is deducted from zero. **Note.** Adjoining boxes may be generated by elements that are not related as siblings or ancestors.
- Vertical margins between a floated [p. 121] box and any other box do not collapse (not even between a float and its in-flow children).
- Vertical margins of elements with 'overflow' other than 'visible' do not collapse with their in-flow children.
- Margins of absolutely [p. 129] positioned boxes do not collapse (not even with their in-flow children).
- If the top and bottom margins of a box are adjacent, then it is possible for margins to collapse through it. In this case, the position of the element depends on its relationship with the other elements whose margins are being collapsed.
  - If the element's margins are collapsed with its parent's top margin, the top border edge of the box is defined to be the same as the parent's.
  - Otherwise, either the element's parent is not taking part in the margin collapsing, or only the parent's bottom margin is involved. The position of the element's top border edge is the same as it would have been if the element had a non-zero top border.

  Note that the positions of elements that have been collapsed through have no effect on the positions of the other elements with whose margins they are being collapsed; the top border edge position is only required for laying out descendants of these elements.

The bottom margin of an in-flow block-level element is always adjoining to the top margin of its next in-flow block-level sibling, unless that sibling has clearance. [p. 128]

The top margin of an in-flow block-level element is adjoining to its first in-flow block-level child's top margin if the element has no top border, no top padding, and the child has no clearance. [p. 128]

The bottom margin of an in-flow block-level element with a 'height' of 'auto' and 'min-height' less than the element's used height is adjoining to its last in-flow block-level child's bottom margin if the element has no bottom padding or border.

An element's own margins are adjoining if the 'min-height' property is zero, and it has neither vertical borders nor vertical padding, and it has a 'height' of either 0 or 'auto', and it does not contain a line box, and all of its in-flow children's margins (if any) are adjoining.

Collapsing is based on the *computed value* of 'padding', 'margin', and 'border'. The collapsed margin is calculated over the computed value of the various margins.

Please consult the examples of margin, padding, and borders [p. 93] for an illustration of collapsed margins.

8.4 Padding properties: 'padding-top', 'padding-right', 'padding-bottom', 'padding-left', and 'padding'

The padding properties specify the width of the padding area [p. 91] of a box. The 'padding' shorthand property sets the padding for all four sides while the other padding properties only set their respective side.

The properties defined in this section refer to the **<padding-width>** value type, which may take one of the following values:

<length>
: Specifies a fixed width.

<percentage>
: The percentage is calculated with respect to the *width* of the generated box's containing block [p. 108] , even for 'padding-top' and 'padding-bottom'. If the containing block's width depends on this element, then the resulting layout is undefined in CSS 2.1.

Unlike margin properties, values for padding values cannot be negative. Like margin properties, percentage values for padding properties refer to the width of the generated box's containing block.

'padding-top', **'padding-right'**, **'padding-bottom'**, **'padding-left'**

Value:	<padding-width> I inherit
Initial:	0
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length

These properties set the top, right, bottom, and left padding of a box.

Example(s):

```
blockquote { padding-top: 0.3em }
```

'padding'

Value:	<padding-width>{1,4} I inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	see individual properties

The 'padding' property is a shorthand property for setting 'padding-top', 'padding-right', 'padding-bottom', and 'padding-left' at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom paddings are set to the first value and the right and left paddings are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.

The surface color or image of the padding area is specified via the 'background' property:

Example(s):

```
h1 {
  background: white;
  padding: 1em 2em;
}
```

The example above specifies a '1em' vertical padding ('padding-top' and 'padding-bottom') and a '2em' horizontal padding ('padding-right' and 'padding-left'). The 'em' unit is relative [p. 48] to the element's font size: '1em' is equal to the size of the font in use.

8.5 Border properties

The border properties specify the width, color, and style of the border area [p. 91] of a box. These properties apply to all elements. Conforming HTML user agents [p. 34] may ignore the border properties on the HTML element.

Note. *Notably for HTML, user agents may render borders for certain elements (e.g., buttons, menus, etc.) differently than for "ordinary" elements.*

8.5.1 Border width: 'border-top-width', 'border-right-width', 'border-bottom-width', 'border-left-width', and 'border-width'

The border width properties specify the width of the border area [p. 91] . The properties defined in this section refer to the **<border-width>** value type, which may take one of the following values:

thin
: A thin border.

medium
: A medium border.

thick
: A thick border.

<length>
: The border's thickness has an explicit value. Explicit border widths cannot be negative.

The interpretation of the first three values depends on the user agent. The following relationships must hold, however:

'thin' <='medium' <= 'thick'.

Furthermore, these widths must be constant throughout a document.

'border-top-width', **'border-right-width'**, **'border-bottom-width'**, **'border-left-width'**

Value:	<border-width> I inherit
Initial:	medium
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	absolute length; '0' if the border style is 'none' or 'hidden'

These properties set the width of the top, right, bottom, and left border of a box.

'border-width'

Value:	<border-width>{1,4}	inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

This property is a shorthand property for setting 'border-top-width', 'border-right-width', 'border-bottom-width', and 'border-left-width' at the same place in the style sheet.

If there is only one value, it applies to all sides. If there are two values, the top and bottom borders are set to the first value and the right and left are set to the second. If there are three values, the top is set to the first value, the left and right are set to the second, and the bottom is set to the third. If there are four values, they apply to the top, right, bottom, and left, respectively.

Example(s):

In the examples below, the comments indicate the resulting widths of the top, right, bottom, and left borders:

```
h1 { border-width: thin }                  /* thin thin thin thin */
h1 { border-width: thin thick }            /* thin thick thin thick */
h1 { border-width: thin thick medium }     /* thin thick medium thick */
```

8.5.2 Border color: 'border-top-color', 'border-right-color', 'border-bottom-color', 'border-left-color', and 'border-color'

The border color properties specify the color of a box's border.

'border-top-color', **'border-right-color'**, **'border-bottom-color'**, **'border-left-color'**

Value:	<color>	transparent	inherit
Initial:	the value of the 'color' property
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	when taken from the 'color' property, the computed value of 'color'; otherwise, as specified

'border-color'

Value:	[<color>	transparent]{1,4}	inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

The 'border-color' property sets the color of the four borders. Values have the following meanings:

<color>
Specifies a color value.
transparent
The border is transparent (though it may have width).

The 'border-color' property can have from one to four values, and the values are set on the different sides as for 'border-width'.

If an element's border color is not specified with a border property, user agents must use the value of the element's 'color' property as the computed value [p. 80] for the border color.

Example(s):

In this example, the border will be a solid black line.

```
p {
  color: black;
  background: white;
  border: solid;
}
```

8.5.3 Border style: 'border-top-style', 'border-right-style', 'border-bottom-style', 'border-left-style', and 'border-style'

The border style properties specify the line style of a box's border (solid, double, dashed, etc.). The properties defined in this section refer to the **<border-style>** value type, which make take one of the following values:

none
No border.
hidden
Same as 'none', except in terms of border conflict resolution [p. 255] for table elements [p. 235] .
dotted
The border is a series of dots.

dashed
The border is a series of short line segments.
solid
The border is a single line segment.
double
The border is two solid lines. The sum of the two lines and the space between them equals the value of 'border-width'.
groove
The border looks as though it were carved into the canvas.
ridge
The opposite of 'groove': the border looks as though it were coming out of the canvas.
inset
The border makes the box look as though it were embedded in the canvas.
outset
The opposite of 'inset': the border makes the box look as though it were coming out of the canvas.

All borders are drawn on top of the box's background. The color of borders drawn for values of 'groove', 'ridge', 'inset', and 'outset' depends on the element's border color properties [p. 101] , but UAs may choose their own algorithm to calculate the actual colors used. For instance, if the 'border-color' has the value 'silver', then a UA could use a gradient of colors from white to dark gray to indicate a sloping border.

'border-top-style', **'border-right-style'**, **'border-bottom-style'**, **'border-left-style'**

Value:	<border-style> I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

'border-style'

Value:	<border-style>{1,4} I inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

The 'border-style' property sets the style of the four borders. It can have from one to four values, and the values are set on the different sides as for 'border-width' above.

Example(s):

```
#xy34 { border-style: solid dotted }
```

In the above example, the horizontal borders will be 'solid' and the vertical borders will be 'dotted'.

Since the initial value of the border styles is 'none', no borders will be visible unless the border style is set.

8.5.4 Border shorthand properties: 'border-top', 'border-bottom', 'border-right', 'border-left', and 'border'

'border-top', **'border-right'**, **'border-bottom'**, **'border-left'**

Value:	[<border-width> II <border-style> II <'border-top-color'>] I inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

This is a shorthand property for setting the width, style, and color of the top, right, bottom, and left border of a box.

Example(s):

```
h1 { border-bottom: thick solid red }
```

The above rule will set the width, style, and color of the border **below** the H1 element. Omitted values are set to their initial values [p. 19] . Since the following rule does not specify a border color, the border will have the color specified by the 'color' property:

```
H1 { border-bottom: thick solid }
```

'border'

Value:	[<border-width> || <border-style> || <'border-top-color'>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

The 'border' property is a shorthand property for setting the same width, color, and style for all four borders of a box. Unlike the shorthand 'margin' and 'padding' properties, the 'border' property cannot set different values on the four borders. To do so, one or more of the other border properties must be used.

Example(s):

For example, the first rule below is equivalent to the set of four rules shown after it:

```
p { border: solid red }
p {
  border-top: solid red;
  border-right: solid red;
  border-bottom: solid red;
  border-left: solid red
}
```

Since, to some extent, the properties have overlapping functionality, the order in which the rules are specified is important.

Example(s):

Consider this example:

```
blockquote {
  border: solid red;
  border-left: double;
  color: black;
}
```

In the above example, the color of the left border is black, while the other borders are red. This is due to 'border-left' setting the width, style, and color. Since the color value is not given by the 'border-left' property, it will be taken from the 'color' property. The fact that the 'color' property is set after the 'border-left' property is not relevant.

8.6 The box model for inline elements in bidi context

For each line box, UAs must take the inline boxes generated for each element and render the margins, borders and padding in visual order (not logical order).

When the element's 'direction' property is 'ltr', the left-most generated box of the first line box in which the element appears has a left margin, left border and left padding, and the right-most generated box of the last line box in which the element appears has a right padding, right border and right margin.

When the element's 'direction' property is 'rtl', the right-most generated box of the first line box in which the element appears has a right padding, right border and right margin, and the left-most generated box of the last line box in which the element appears has a left margin, left border and left padding.

9 Visual formatting model

Contents

9.1 Introduction to the visual formatting model

This chapter and the next describe the visual formatting model: how user agents process the document tree [p. 33] for visual media [p. 87] .

In the visual formatting model, each element in the document tree generates zero or more boxes according to the box model [p. 91] . The layout of these boxes is governed by:

- box dimensions [p. 91] and type [p. 109] .
- positioning scheme [p. 114] (normal flow, float, and absolute positioning).
- relationships between elements in the document tree. [p. 33]
- external information (e.g., viewport size, intrinsic [p. 32] dimensions of images, etc.).

The properties defined in this chapter and the next apply to both continuous media [p. 89] and paged media [p. 89] . However, the meanings of the margin properties [p. 95] vary when applied to paged media (see the page model [p. 195] for details).

The visual formatting model does not specify all aspects of formatting (e.g., it does not specify a letter-spacing algorithm). Conforming user agents [p. 34] may behave differently for those formatting issues not covered by this specification.

9.1.1 The viewport

User agents for continuous media [p. 89] generally offer users a *viewport* (a window or other viewing area on the screen) through which users consult a document. User agents may change the document's layout when the viewport is resized (see the initial containing block [p. 149]).

When the viewport is smaller than the area of the canvas on which the document is rendered, the user agent should offer a scrolling mechanism. There is at most one viewport per canvas [p. 28] , but user agents may render to more than one canvas (i.e., provide different views of the same document).

9.1.2 Containing blocks

In CSS 2.1, many box positions and sizes are calculated with respect to the edges of a rectangular box called a *containing block*. In general, generated boxes act as containing blocks for descendant boxes; we say that a box "establishes" the containing block for its descendants. The phrase "a box's containing block" means "the containing block in which the box lives," not the one it generates.

Each box is given a position with respect to its containing block, but it is not confined by this containing block; it may overflow [p. 169] .

User agents may treat float as 'none' and/or position as 'static' on the root element.

The details [p. 149] of how a containing block's dimensions are calculated are described in the next chapter [p. 149] .

9.2 Controlling box generation

The following sections describe the types of boxes that may be generated in CSS 2.1. A box's type affects, in part, its behavior in the visual formatting model. The 'display' property, described below, specifies a box's type.

9.2.1 Block-level elements and block boxes

Block-level elements are those elements of the source document that are formatted visually as blocks (e.g., paragraphs). Several values of the 'display' property make an element block-level: 'block', 'list-item', and 'run-in' (part of the time; see run-in boxes [p. 111]), and 'table'.

Block-level elements generate a *principal block box* that contains either only *block boxes* or only inline boxes [p. 111] . The principal block box establishes the containing block [p. 108] for descendant boxes and generated content and is also the box involved in any positioning scheme. Principal block boxes participate in a block formatting context [p. 117] .

Some block-level elements generate additional boxes outside of the principal box: 'list-item' elements. These additional boxes are placed with respect to the principal box.

Anonymous block boxes

In a document like this:

```
<DIV>
  Some text
  <P>More text
</DIV>
```

(and assuming the DIV and the P both have 'display: block'), the DIV appears to have both inline content and block content. To make it easier to define the formatting, we assume that there is an *anonymous block box* around "Some text".

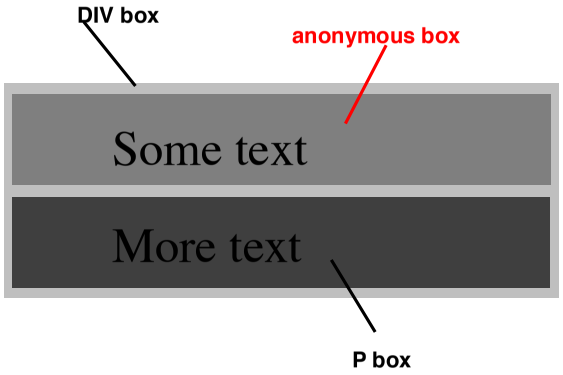


Diagram showing the three boxes, of which one is anonymous, for the example above.

In other words: if a block box (such as that generated for the DIV above) has another block box inside it (such as the P above), then we force it to have *only* block boxes inside it, by wrapping any inline boxes in an anonymous block box.

When an inline box contains a block box, the inline box (and its inline ancestors within the same line box) are broken around the block. The line boxes before the break and after the break are enclosed in anonymous boxes, and the block box becomes a sibling of those anonymous boxes.

Example(s):

This model would apply in the following example if the following rules:

```
body { display: inline }
p    { display: block }
```

were used with this HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HEAD>
<TITLE>Anonymous text interrupted by a block</TITLE>
</HEAD>
<BODY>
This is anonymous text before the P.
<P>This is the content of P.</P>
This is anonymous text after the P.
</BODY>
```

The BODY element contains a chunk (C1) of anonymous text followed by a block-level element followed by another chunk (C2) of anonymous text. The resulting boxes would be an anonymous block box for BODY, containing an anonymous block box around C1, the P block box, and another anonymous block box around C2.

The properties of anonymous boxes are inherited from the enclosing non-anonymous box (in the example: the one for DIV). Non-inherited properties have their initial value. For example, the font of the anonymous box is inherited from the DIV, but the margins will be 0.

Properties set on elements that are turned into anonymous block boxes still apply to the content of the element. For example, if a border had been set on the BODY element in the above example, the border would be drawn around C1 and C2.

9.2.2 Inline-level elements and inline boxes

Inline-level elements are those elements of the source document that do not form new blocks of content; the content is distributed in lines (e.g., emphasized pieces of text within a paragraph, inline images, etc.). Several values of the 'display' property make an element inline: 'inline', 'inline-table', and 'run-in' (part of the time; see run-in boxes [p. 111]). Inline-level elements generate *inline boxes*.

Anonymous inline boxes

In a document with HTML markup like this:

```
<p>Some <em>emphasized</em> text</p>
```

The <p> generates a block box, with several inline boxes inside it. The box for "emphasized" is an inline box generated by an inline element (), but the other boxes ("Some" and "text") are inline boxes generated by a block-level element (<p>). The latter are called anonymous inline boxes, because they don't have an associated inline-level element.

Such anonymous inline boxes inherit inheritable properties from their block parent box. Non-inherited properties have their initial value. In the example, the color of the anonymous inline boxes is inherited from the P, but the background is transparent.

Whitespace content that would subsequently be collapsed away according to the 'white-space' property does not generate any anonymous inline boxes.

If it is clear from the context which type of anonymous box is meant, both anonymous inline boxes and anonymous block boxes are simply called anonymous boxes in this specification.

There are more types of anonymous boxes that arise when formatting tables [p. 238] .

9.2.3 Run-in boxes

A *run-in box* behaves as follows:

1. If the run-in box contains a block [p. 109] box, the run-in box becomes a block box.
2. If a block [p. 109] box (that does not float and is not absolutely positioned [p. 129]) follows the run-in box, the run-in box becomes the first inline box of the block box.
3. Otherwise, the run-in box becomes a block box.

A 'run-in' box is useful for run-in headers, as in this example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>A run-in box example</TITLE>
    <STYLE type="text/css">
      H3 { display: run-in }
    </STYLE>
  </HEAD>
  <BODY>
    <H3>A run-in heading.</H3>
    <P>And a paragraph of text that
       follows it.
  </BODY>
</HTML>
```

This example might be formatted as:

A run-in heading. And a
paragraph of text that
follows it.

Despite appearing visually part of the following block box, a run-in element still inherits properties from its parent in the source tree.

Please consult the section on generated content [p. ??] for information about how run-in boxes interact with generated content.

9.2.4 The 'display' property

'display'

Value:	inline | block | list-item | run-in | inline-block | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none | inherit
Initial:	inline
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all
Computed value:	see text

The values of this property have the following meanings:

block
This value causes an element to generate a block box.
inline-block
This value causes an element to generate a block box, which itself is flowed as a single inline box, similar to a replaced element. The inside of an inline-block is formatted as a block box, and the element itself is formatted as a replaced element on the line.
inline
This value causes an element to generate one or more inline boxes.
list-item
This value causes an element (e.g., LI in HTML) to generate a principal block box and a list-item inline box. For information about lists and examples of list formatting, please consult the section on lists [p. 188] .
none
This value causes an element to generate **no** boxes in the formatting structure [p. 28] (i.e., the element has no effect on layout). Descendant elements do not generate any boxes either; this behavior **cannot** be overridden by setting the 'display' property on the descendants.

Please note that a display of 'none' does not create an invisible box; it creates no box at all. CSS includes mechanisms that enable an element to generate boxes in the formatting structure that affect formatting but are not visible themselves. Please consult the section on visibility [p. 174] for details.

run-in
This value creates either block or inline boxes, depending on context. Properties apply to run-in boxes based on their final status (inline-level or block-level).
table, **inline-table**, **table-row-group**, **table-column**, **table-column-group**, **table-header-group**, **table-footer-group**, **table-row**, **table-cell**, and **table-caption**
These values cause an element to behave like a table element (subject to restrictions described in the chapter on tables [p. 235]).

The computed value is the same as the specified value, except for positioned and floating elements (see Relationships between 'display', 'position', and 'float' [p. 131]) and for the root element. For the root element, the computed value is as follows: 'inline-table' and 'table' become 'table', 'none' stays 'none', everything else becomes 'block'.

Note that although the initial value [p. 19] of 'display' is 'inline', rules in the user agent's default style sheet [p. 82] may override [p. 79] this value. See the sample style sheet [p. 293] for HTML 4.0 in the appendix.

Example(s):

Here are some examples of the 'display' property:

```
p   { display: block }
em  { display: inline }
li  { display: list-item }
img { display: none }      /* Don't display images */
```


9.3 Positioning schemes

In CSS 2.1, a box may be laid out according to three *positioning schemes:*

1. Normal flow [p. 117] . In CSS 2.1, normal flow includes block formatting [p. 117] of block [p. 109] boxes, inline formatting [p. 118] of inline [p. 111] boxes, relative positioning [p. 120] of block or inline boxes, and positioning of run-in [p. 111] boxes.
2. Floats [p. 121] . In the float model, a box is first laid out according to the normal flow, then taken out of the flow and shifted to the left or right as far as possible. Content may flow along the side of a float.
3. Absolute positioning [p. 129] . In the absolute positioning model, a box is removed from the normal flow entirely (it has no impact on later siblings) and assigned a position with respect to a containing block.

Note. *CSS 2.1's positioning schemes help authors make their documents more accessible by allowing them to avoid mark-up tricks (e.g., invisible images) used for layout effects.*

9.3.1 Choosing a positioning scheme: 'position' property

The 'position' and 'float' properties determine which of the CSS 2.1 positioning algorithms is used to calculate the position of a box.

'position'

Value:	static I relative I absolute I fixed I inherit
Initial:	static
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

The values of this property have the following meanings:

static
The box is a normal box, laid out according to the normal flow [p. 117] . The 'top', 'right', 'bottom', and 'left' properties do not apply.
relative
The box's position is calculated according to the normal flow [p. 117] (this is called the position in normal flow). Then the box is offset relative [p. 120] to its normal position. When a box B is relatively positioned, the position of the following box is calculated as though B were not offset.
absolute
The box's position (and possibly size) is specified with the 'top', 'right', 'bottom', and 'left' properties. These properties specify offsets with respect to the box's

containing block [p. 108] . Absolutely positioned boxes are taken out of the normal flow. This means they have no impact on the layout of later siblings. Also, though absolutely positioned [p. 129] boxes have margins, they do not collapse [p. 97] with any other margins.

fixed

The box's position is calculated according to the 'absolute' model, but in addition, the box is fixed [p. 129] with respect to some reference. As with the 'absolute' model, the box's margins do not collapse with any other margins. In the case of handheld, projection, screen, tty, and tv media types, the box is fixed with respect to the viewport [p. 108] and doesn't move when scrolled. In the case of the print media type, the box is fixed with respect to the page, even if that page is seen through a viewport [p. 108] (in the case of a print-preview, for example). For other media types, the presentation is undefined. Authors may wish to specify 'fixed' in a media-dependent way. For instance, an author may want a box to remain at the top of the viewport [p. 108] on the screen, but not at the top of each printed page. The two specifications may be separated by using an @media rule [p. 88] , as in:

Example(s):

```
@media screen {
  h1#first { position: fixed }
}
@media print {
  h1#first { position: static }
}
```

9.3.2 Box offsets: 'top', 'right', 'bottom', 'left'

An element is said to be *positioned* if its 'position' property has a value other than 'static'. Positioned elements generate positioned boxes, laid out according to four properties:

'top'

Value:	<length> I <percentage> I auto I inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual
Computed value:	for 'position:relative', see section Relative Positioning. [p. 120] For 'position:static', 'auto'. Otherwise: if specified as a length, the corresponding absolute length; if specified as a percentage, the specified value; otherwise, 'auto'.

This property specifies how far an absolutely positioned [p. 129] box's top margin edge is offset below the top edge of the box's containing block [p. 108] . For relatively positioned boxes, the offset is with respect to the top edges of the box itself (i.e., the box is given a position in the normal flow, then offset from that position according to these properties). Note: For absolutely positioned elements whose containing block is based on a block-level element, this property is an offset from the *padding* edge of that element.

'right'

Value:	<length> I <percentage> I auto I inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	for 'position:relative', see section Relative Positioning. [p. 120] For 'position:static', 'auto'. Otherwise: if specified as a length, the corresponding absolute length; if specified as a percentage, the specified value; otherwise, 'auto'.

Like 'top', but specifies how far a box's right margin edge is offset to the left of the right edge of the box's containing block [p. 108] . For relatively positioned boxes, the offset is with respect to the right edge of the box itself. Note: For absolutely positioned elements whose containing block is based on a block-level element, this property is an offset from the *padding* edge of that element.

'bottom'

Value:	<length> I <percentage> I auto I inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to height of containing block
Media:	visual
Computed value:	for 'position:relative', see section Relative Positioning. [p. 120] For 'position:static', 'auto'. Otherwise: if specified as a length, the corresponding absolute length; if specified as a percentage, the specified value; otherwise, 'auto'.

Like 'top', but specifies how far a box's bottom margin edge is offset above the bottom of the box's containing block [p. 108] . For relatively positioned boxes, the offset is with respect to the bottom edge of the box itself. Note: For absolutely positioned elements whose containing block is based on a block-level element, this property is an offset from the *padding* edge of that element.

'left'

Value:	<length> I <percentage> I auto I inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	for 'position:relative', see section Relative Positioning. [p. 120] For 'position:static', 'auto'. Otherwise: if specified as a length, the corresponding absolute length; if specified as a percentage, the specified value; otherwise, 'auto'.

Like 'top', but specifies how far a box's left margin edge is offset to the right of the left edge of the box's containing block [p. 108] . For relatively positioned boxes, the offset is with respect to the left edge of the box itself. Note: For absolutely positioned elements whose containing block is based on a block-level element, this property is an offset from the *padding* edge of that element.

The values for the four properties have the following meanings:

<length>
The offset is a fixed distance from the reference edge. Negative values are allowed.

<percentage>
The offset is a percentage of the containing block's box width (for 'left' or 'right') or height (for 'top' and 'bottom'). For 'top' and 'bottom', if the height of the containing block is not specified explicitly (i.e., it depends on content height), the percentage value is interpreted like 'auto'. Negative values are allowed.

auto
The effect of this value depends on which of related properties have the value 'auto' as well. See the sections on the width [p. 154] and height [p. 161] of absolutely positioned [p. 129] , non-replaced elements for details.

9.4 Normal flow

Boxes in the normal flow belong to a formatting context, which may be block or inline, but not both simultaneously. Block [p. 109] boxes participate in a block formatting [p. 117] context. Inline boxes [p. 111] participate in an inline formatting [p. 118] context.

9.4.1 Block formatting context

Floats, absolutely positioned elements, inline-blocks, table-cells, and elements with 'overflow' other than 'visible' establish new block formatting contexts.

In a block formatting context, boxes are laid out one after the other, vertically, beginning at the top of a containing block. The vertical distance between two sibling boxes is determined by the 'margin' properties. Vertical margins between adjacent block boxes in a block formatting context collapse [p. 97] .

In a block formatting context, each box's left outer edge touches the left edge of the containing block (for right-to-left formatting, right edges touch). This is true even in the presence of floats (although a box's *line boxes* may shrink due to the floats).

For information about page breaks in paged media, please consult the section on allowed page breaks [p. 201] .

9.4.2 Inline formatting context

In an inline formatting context, boxes are laid out horizontally, one after the other, beginning at the top of a containing block. Horizontal margins, borders, and padding are respected between these boxes. The boxes may be aligned vertically in different ways: their bottoms or tops may be aligned, or the baselines of text within them may be aligned. The rectangular area that contains the boxes that form a line is called a *line box*.

The width of a line box is determined by a containing block [p. 108] and the presence of floats. The height of a line box is determined by the rules given in the section on line height calculations [p. 164] .

A line box is always tall enough for all of the boxes it contains. However, it may be taller than the tallest box it contains (if, for example, boxes are aligned so that baselines line up). When the height of a box B is less than the height of the line box containing it, the vertical alignment of B within the line box is determined by the 'vertical-align' property. When several inline boxes cannot fit horizontally within a single line box, they are distributed among two or more vertically-stacked line boxes. Thus, a paragraph is a vertical stack of line boxes. Line boxes are stacked with no vertical separation and they never overlap.

In general, the left edge of a line box touches the left edge of its containing block and the right edge touches the right edge of its containing block. However, floating boxes may come between the containing block edge and the line box edge. Thus, although line boxes in the same inline formatting context generally have the same width (that of the containing block), they may vary in width if available horizontal space is reduced due to floats [p. 121] . Line boxes in the same inline formatting context generally vary in height (e.g., one line might contain a tall image while the others contain only text).

When the total width of the inline boxes on a line is less than the width of the line box containing them, their horizontal distribution within the line box is determined by the 'text-align' property. If that property has the value 'justify', the user agent may stretch the inline boxes as well.

When an inline box exceeds the width of a line box, it is split into several boxes and these boxes are distributed across several line boxes. If an inline box cannot be split (e.g. if the inline box contains a single character, or language specific word breaking rules disallow a break within the inline box, or if the inline box is affected by a white-space value of nowrap or pre), then the inline box overflows the line box.

When an inline box is split, margins, borders, and padding have no visual effect where the split occurs (or at any split, when there are several).

Inline boxes may also be split into several boxes *within the same line box* due to bidirectional text processing [p. 144] .

Here is an example of inline box construction. The following paragraph (created by the HTML block-level element P) contains anonymous text interspersed with the elements EM and STRONG:

```
<P>Several <EM>emphasized words</EM> appear
<STRONG>in this</STRONG> sentence, dear.</P>
```

The P element generates a block box that contains five inline boxes, three of which are anonymous:

- Anonymous: "Several"
- EM: "emphasized words"
- Anonymous: "appear"
- STRONG: "in this"
- Anonymous: "sentence, dear."

To format the paragraph, the user agent flows the five boxes into line boxes. In this example, the box generated for the P element establishes the containing block for the line boxes. If the containing block is sufficiently wide, all the inline boxes will fit into a single line box:

```
Several emphasized words appear in this sentence, dear.
```

If not, the inline boxes will be split up and distributed across several line boxes. The previous paragraph might be split as follows:

```
Several emphasized words appear
in this sentence, dear.
```

or like this:

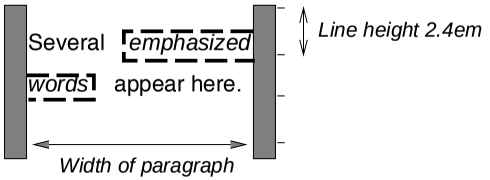
```
Several emphasized
words appear in this
sentence, dear.
```

In the previous example, the EM box was split into two EM boxes (call them "split1" and "split2"). Margins, borders, padding, or text decorations have no visible effect after split1 or before split2.

Consider the following example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Example of inline flow on several lines</TITLE>
    <STYLE type="text/css">
      EM {
        padding: 2px;
        margin: 1em;
        border-width: medium;
        border-style: dashed;
        line-height: 2.4em;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <P>Several <EM>emphasized words</EM> appear here.</P>
  </BODY>
</HTML>
```

Depending on the width of the P, the boxes may be distributed as follows:



- The margin is inserted before "emphasized" and after "words".
- The padding is inserted before, above, and below "emphasized" and after, above, and below "words". A dashed border is rendered on three sides in each case.

9.4.3 Relative positioning

Once a box has been laid out according to the normal flow [p. 117] or floated, it may be shifted relative to this position. This is called *relative positioning*. Offsetting a box (B1) in this way has no effect on the box (B2) that follows: B2 is given a position as if B1 were not offset and B2 is not re-positioned after B1's offset is applied. This implies that relative positioning may cause boxes to overlap.

A relatively positioned box keeps its normal flow size, including line breaks and the space originally reserved for it. The section on containing blocks [p. 108] explains when a relatively positioned box establishes a new containing block.

For relatively positioned elements, 'left' and 'right' move the box(es) horizontally, without changing their size. 'left' moves the boxes to the right, and 'right' moves them to the left. Since boxes are not split or stretched as a result of 'left' or 'right', the computed values are always: left = -right.

If both 'left' and 'right' are 'auto' (their initial values), the computed values are '0' (i.e., the boxes stay in their original position).

If 'left' is 'auto', its computed value is minus the value of 'right' (i.e., the boxes move to the left by the value of 'right').

If 'right' is specified as 'auto', its computed value is minus the value of 'left'.

If neither 'left' nor 'right' is 'auto', the position is over-constrained, and one of them has to be ignored. If the 'direction' property is 'ltr', the value of 'left' wins and 'right' becomes -'left'. If 'direction' is 'rtl', 'right' wins and 'left' is ignored.

Example(s):

Example. The following three rules are equivalent:

```
div.a8 { position: relative; direction: ltr; left: -1em; right: auto }
div.a8 { position: relative; direction: ltr; left: auto; right: 1em }
div.a8 { position: relative; direction: ltr; left: -1em; right: 5em }
```

The 'top' and 'bottom' properties move relatively positioned element(s) up or down without changing their size. 'top' moves the boxes down, and 'bottom' moves them up. Since boxes are not split or stretched as a result of 'top' or 'bottom', the computed values are always: top = -bottom. If both are 'auto', their computed values are both '0'. If one of them is 'auto', it becomes the negative of the other. If neither is 'auto', 'bottom' is ignored (i.e., the computed value of 'bottom' will be minus the value of 'top').

Dynamic movement of relatively positioned boxes can produce animation effects in scripting environments (see also the 'visibility' property). Relative positioning may also be used as a general form of superscripting and subscripting except that line height is not automatically adjusted to take the positioning into consideration. See the description of line height calculations [p. 164] for more information.

Examples of relative positioning are provided in the section comparing normal flow, floats, and absolute positioning [p. 132] .

9.5 Floats

A float is a box that is shifted to the left or right on the current line. The most interesting characteristic of a float (or "floated" or "floating" box) is that content may flow along its side (or be prohibited from doing so by the 'clear' property). Content flows down the right side of a left-floated box and down the left side of a right-floated box. The following is an introduction to float positioning and content flow; the exact rules [p. 127] governing float behavior are given in the description of the 'float' property.

A floated box is shifted to the left or right until its outer edge touches the containing block edge or the outer edge of another float. The top of the floated box is aligned with the top of the current line box (or bottom of the preceding block box if no line box exists).

If there isn't enough horizontal room for the float, it is shifted downward until either it fits or there are no more floats present.

Since a float is not in the flow, non-positioned block boxes created before and after the float box flow vertically as if the float didn't exist. However, line boxes created next to the float are shortened to make room for the floated box. If a shortened line box is too small to contain any further content, then it is shifted downward until either it fits or there are no more floats present. Any content in the current line before a floated box is reflowed in the first available line on the other side of the float. In other words, if inline boxes are placed on the line before a left float is encountered that fits in the remaining line box space, the left float is placed on that line, aligned with the top of the line box, and then the inline boxes already on the line are moved accordingly to the right of the float (the right being the other side of the left float) and vice versa for rtl and right floats.

The margin box of an element in the normal flow that establishes a new block formatting context (such as a table, or element with 'overflow' other than 'visible') must not overlap any floats in the same block formatting context as the element itself. If necessary, implementations should clear the said element by placing it below any preceding floats, but may place it adjacent to such floats if there is sufficient space.

Example(s):

Example. In the following document fragment, the containing block is too short to contain the content, so the content gets moved to below the floats where it is aligned in the line box according to the text-align property.

```
p { width: 10em; border: solid aqua; }
span { float: left; width: 5em; height: 5em; border: solid blue; }

...

<p>
  <span> </span>
  Supercalifragilisticexpialidocious
</p>
```

This fragment might look like this:

Several floats may be adjacent, and this model also applies to adjacent floats in the same line.

Example(s):

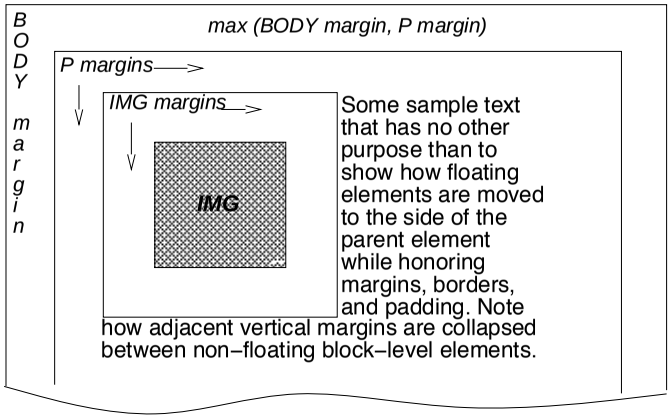
The following rule floats all IMG boxes with `class="icon"` to the left (and sets the left margin to '0'):

```
img.icon {
  float: left;
  margin-left: 0;
}
```

Consider the following HTML source and style sheet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Float example</TITLE>
    <STYLE type="text/css">
      IMG { float: left }
      BODY, P, IMG { margin: 2em }
    </STYLE>
  </HEAD>
  <BODY>
    <P><IMG src=img.png alt="This image will illustrate floats">
       Some sample text that has no other...
  </BODY>
</HTML>
```

The IMG box is floated to the left. The content that follows is formatted to the right of the float, starting on the same line as the float. The line boxes to the right of the float are shortened due to the float's presence, but resume their "normal" width (that of the containing block established by the P element) after the float. This document might be formatted as:



Formatting would have been exactly the same if the document had been:

```
<BODY>
  <P>Some sample text
  <IMG src=img.png alt="This image will illustrate floats">
           that has no other...
</BODY>
```

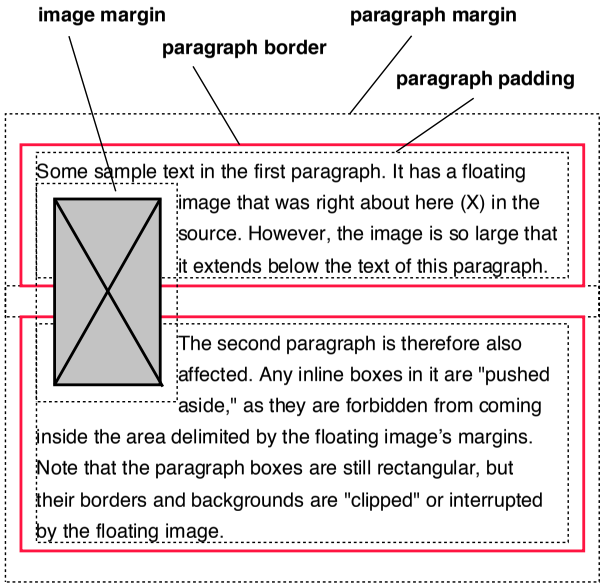
because the content to the left of the float is displaced by the float and reflowed down its right side.

As stated in section 8.3.1 [p. 97] , the margins of floating boxes never collapse [p. 97] with margins of adjacent boxes. Thus, in the previous example, vertical margins do not collapse [p. 97] between the P box and the floated IMG box.

The contents of floats are stacked as if floats generated new stacking contexts, except that any elements that actually create new stacking contexts take part in the float's parent's stacking context. A float can overlap other boxes in the normal flow (e.g., when a normal flow box next to a float has negative margins). When this happens, floats are rendered in front of non-positioned in-flow blocks, but behind in-flow inlines.

Example(s):

Here is another illustration, showing what happens when a float overlaps borders of elements in the normal flow.



A floating image obscures borders of block boxes it overlaps.

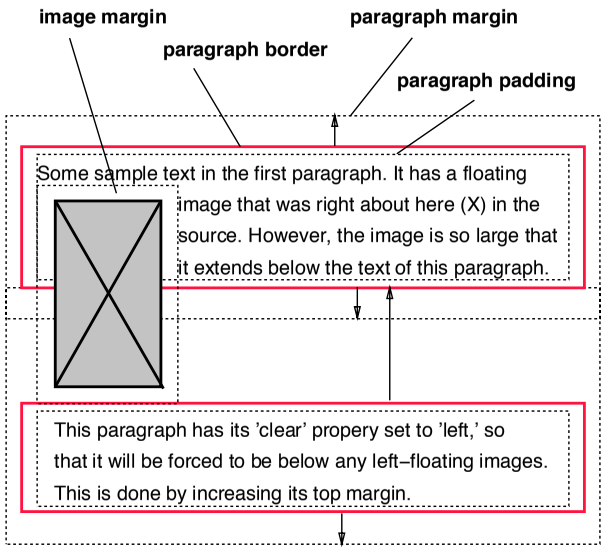
The following example illustrates the use of the 'clear' property to prevent content from flowing next to a float.

Example(s):

Assuming a rule such as this:

```
p { clear: left }
```

formatting might look like this:



Both paragraphs have set 'clear: left', which causes the second paragraph to be "pushed down" to a position below the float — its top margin expands to accomplish this (see the 'clear' property).

9.5.1 Positioning the float: the 'float' property

'float'

Value:	left I right I none I inherit
Initial:	none
Applies to:	all, but see 9.7 [p. 131]
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies whether a box should float to the left, right, or not at all. It may be set for elements that generate boxes that are not absolutely positioned [p. 129] . The values of this property have the following meanings:

left

The element generates a block [p. 109] box that is floated to the left. Content flows on the right side of the box, starting at the top (subject to the 'clear' property).

right
Same as 'left', but content flows on the left side of the box, starting at the top.
none
The box is not floated.

Here are the precise rules that govern the behavior of floats:

1. The left outer edge [p. 92] of a left-floating box may not be to the left of the left edge of its containing block [p. 108] . An analogous rule holds for right-floating elements.
2. If the current box is left-floating, and there are any left-floating boxes generated by elements earlier in the source document, then for each such earlier box, either the left outer edge [p. 92] of the current box must be to the right of the right outer edge [p. 92] of the earlier box, or its top must be lower than the bottom of the earlier box. Analogous rules hold for right-floating boxes.
3. The right outer edge [p. 92] of a left-floating box may not be to the right of the left outer edge [p. 92] of any right-floating box that is to the right of it. Analogous rules hold for right-floating elements.
4. A floating box's outer top [p. 92] may not be higher than the top of its containing block [p. 108] .
5. The outer top [p. 92] of a floating box may not be higher than the outer top of any block [p. 109] or floated [p. 121] box generated by an element earlier in the source document.
6. The outer top [p. 92] of an element's floating box may not be higher than the top of any line-box [p. 118] containing a box generated by an element earlier in the source document.
7. A left-floating box that has another left-floating box to its left may not have its right outer edge to the right of its containing block's right edge. (Loosely: a left float may not stick out at the right edge, unless it is already as far to the left as possible.) An analogous rule holds for right-floating elements.
8. A floating box must be placed as high as possible.
9. A left-floating box must be put as far to the left as possible, a right-floating box as far to the right as possible. A higher position is preferred over one that is further to the left/right.

When the rules above do not result in an exact vertical position, as may be the case when the float occurs between two collapsing margins, the float is positioned as if it had an otherwise empty anonymous block parent [p. 109] taking part in the flow. The position of such a parent is defined by the rules [p. 97] in the section on margin collapsing.

References to other elements in these rules refer only to other elements in the same block formatting context [p. 117] as the float..

9.5.2 Controlling flow next to floats: the 'clear' property

'clear'

Value:	none I left I right I both I inherit
Initial:	none
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property indicates which sides of an element's box(es) may *not* be adjacent to an earlier floating box. The 'clear' property does not consider floats inside the element itself or in other block formatting contexts.

For run-in boxes [p. 111] , this property applies to the final block box to which the run-in box belongs.

The *clearance* dimension is introduced as a dimension above the margin-top of an element that is used to push the element vertically (typically downward).

Values have the following meanings when applied to non-floating block boxes:

left
The clearance of the generated box is set to the amount necessary to place the top border edge is below the bottom outer edge of any left-floating boxes that resulted from elements earlier in the source document.
right
The clearance of the generated box is set to the amount necessary to place the top border edge is below the bottom outer edge of any right-floating boxes that resulted from elements earlier in the source document.
both
The clearance of the generated box is set to the amount necessary to place the top border edge is below the bottom outer edge of any right-floating and left-floating boxes that resulted from elements earlier in the source document.
none
No constraint on the box's position with respect to floats.

Computing the clearance of an element on which 'clear' is set is done by first determining the hypothetical position of the element's top border edge within its parent block. This position is determined after the top margin of the element has been collapsed with previous adjacent margins (including the top margin of the parent block).

If the element's top border edge has not passed the relevant floats, then its clearance is set to the amount necessary to place the border edge of the block even with the bottom outer edge of the lowest float that must be cleared.

When the property is set on floating elements, it results in a modification of the rules [p. 127] for positioning the float. An extra constraint (#10) is added:

- The top outer edge [p. 92] of the float must be below the bottom outer edge of all earlier left-floating boxes (in the case of 'clear: left'), or all earlier right-floating boxes (in the case of 'clear: right'), or both ('clear: both').

Note. *This property applied to all elements in CSS1 [p. ??] . Implementations may therefore have supported this property on all elements. In CSS2 and CSS 2.1 the 'clear' property only applies to block-level elements. Therefore authors should only use this property on block-level elements. If an implementation does support clear on inline elements, rather than setting a clearance as explained above, the implementation should force a break and effectively insert one or more empty line boxes (or shifting the new line box downward as described in section 9.5 [p. 121]) to move the top of the cleared inline's line box to below the respective floating box(es).*

Example:

```
span { clear: left }
```

9.6 Absolute positioning

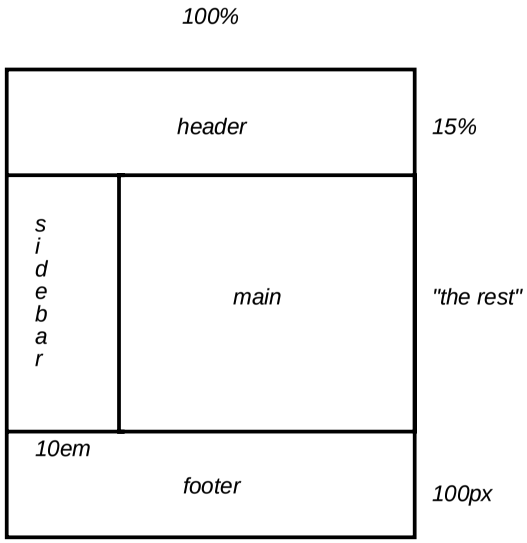
In the absolute positioning model, a box is explicitly offset with respect to its containing block. It is removed from the normal flow entirely (it has no impact on later siblings). An absolutely positioned box establishes a new containing block for normal flow children and absolutely (but not fixed) positioned descendants. However, the contents of an absolutely positioned element do not flow around any other boxes. They may obscure the contents of another box (or be obscured themselves), depending on the stack levels [p. 142] of the overlapping boxes.

References in this specification to an *absolutely positioned element* (or its box) imply that the element's 'position' property has the value 'absolute' or 'fixed'.

9.6.1 Fixed positioning

Fixed positioning is a subcategory of absolute positioning. The only difference is that for a fixed positioned box, the containing block is established by the viewport [p. 108] . For continuous media [p. 89] , fixed boxes do not move when the document is scrolled. In this respect, they are similar to fixed background images [p. 206] . For paged media [p. 195] , boxes with fixed positions are repeated on every page. This is useful for placing, for instance, a signature at the bottom of each page.

Authors may use fixed positioning to create frame-like presentations. Consider the following frame layout:



This might be achieved with the following HTML document and style rules:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>A frame document with CSS 2.1</TITLE>
    <STYLE type="text/css">
      BODY { height: 8.5in } /* Required for percentage heights below */
      #header {
        position: fixed;
        width: 100%;
        height: 15%;
        top: 0;
        right: 0;
        bottom: auto;
        left: 0;
      }
      #sidebar {
        position: fixed;
        width: 10em;
        height: auto;
        top: 15%;
        right: auto;
        bottom: 100px;
        left: 0;
      }
      #main {
        position: fixed;
        width: auto;
        height: auto;
        top: 15%;
```

```
        right: 0;
        bottom: 100px;
        left: 10em;
      }
      #footer {
        position: fixed;
        width: 100%;
        height: 100px;
        top: auto;
        right: 0;
        bottom: 0;
        left: 0;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <DIV id="header"> ...  </DIV>
    <DIV id="sidebar"> ...  </DIV>
    <DIV id="main"> ...  </DIV>
    <DIV id="footer"> ...  </DIV>
  </BODY>
</HTML>
```

9.7 Relationships between 'display', 'position', and 'float'

The three properties that affect box generation and layout — 'display', 'position', and 'float' — interact as follows:

1. If 'display' has the value 'none', then 'position' and 'float' do not apply. In this case, the element generates no box.
2. Otherwise, if 'position' has the value 'absolute' or 'fixed', the box is absolutely positioned, the computed value of 'float' is 'none', and display is set according to this table:

Specified value	Computed value
inline-table	table
inline, run-in, table-row-group, table-column, table-column-group, table-header-group, table-footer-group, table-row, table-cell, table-caption, inline-block	block
others	same as specified

The position of the box will be determined by the 'top', 'right', 'bottom' and 'left' properties and the box's containing block.

3. Otherwise, if 'float' has a value other than 'none', the box is floated and 'display' is set according to this table:

Specified value	Computed value
inline-table	table
inline, run-in, table-row-group, table-column, table-column-group, table-header-group, table-footer-group, table-row, table-cell, table-caption, inline-block	block
others	same as specified

4. Otherwise, the remaining 'display' property values apply as specified.

9.8 Comparison of normal flow, floats, and absolute positioning

To illustrate the differences between normal flow, relative positioning, floats, and absolute positioning, we provide a series of examples based on the following HTML:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Comparison of positioning schemes</TITLE>
  </HEAD>
  <BODY>
    <P>Beginning of body contents.
      <SPAN id="outer"> Start of outer contents.
      <SPAN id="inner"> Inner contents.</SPAN>
      End of outer contents.</SPAN>
      End of body contents.
    </P>
  </BODY>
</HTML>
```

In this document, we assume the following rules:

```
body { display: block; font-size:12px; line-height: 200%;
       width: 400px; height: 400px }
p    { display: block }
span { display: inline }
```

The final positions of boxes generated by the *outer* and *inner* elements vary in each example. In each illustration, the numbers to the left of the illustration indicate the normal flow [p. 117] position of the double-spaced (for clarity) lines.

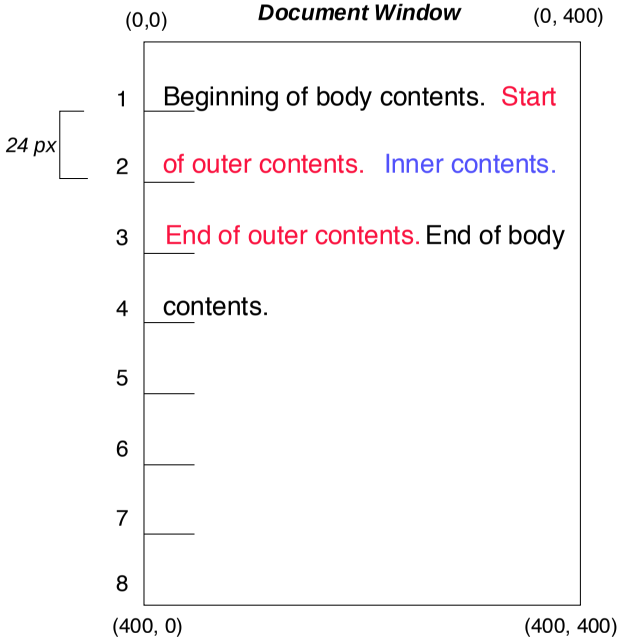
Note. The diagrams in this section are illustrative and not to scale. They are meant to highlight the differences between the various positioning schemes in CSS 2.1, and are not intended to be reference renderings of the examples given.

9.8.1 Normal flow

Consider the following CSS declarations for *outer* and *inner* that don't alter the normal flow [p. 117] of boxes:

```
#outer { color: red }
#inner { color: blue }
```

The P element contains all inline content: anonymous inline text [p. 111] and two SPAN elements. Therefore, all of the content will be laid out in an inline formatting context, within a containing block established by the P element, producing something like:



9.8.2 Relative positioning

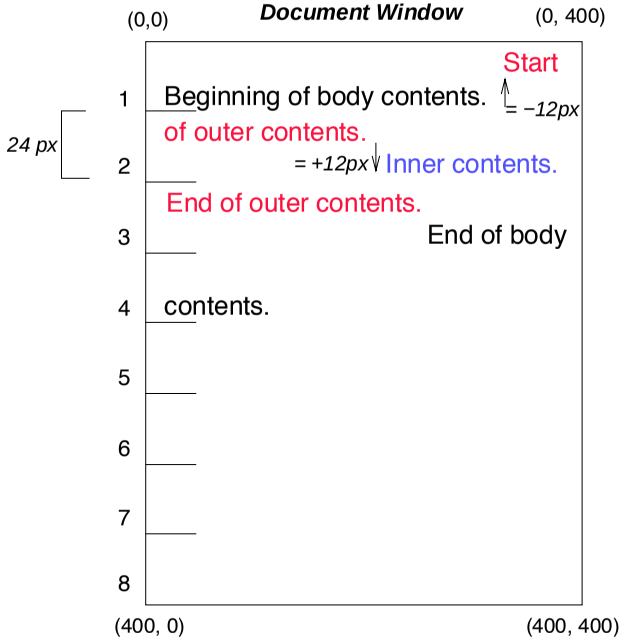
To see the effect of relative positioning [p. 120] , we specify:

```
#outer { position: relative; top: -12px; color: red }
#inner { position: relative; top: 12px; color: blue }
```

Text flows normally up to the *outer* element. The *outer* text is then flowed into its normal flow position and dimensions at the end of line 1. Then, the inline boxes containing the text (distributed over three lines) are shifted as a unit by '-12px' (upwards).

The contents of *inner*, as a child of *outer*, would normally flow immediately after the words "of outer contents" (on line 1.5). However, the *inner* contents are themselves offset relative to the *outer* contents by '12px' (downwards), back to their original position on line 2.

Note that the content following *outer* is not affected by the relative positioning of *outer*.



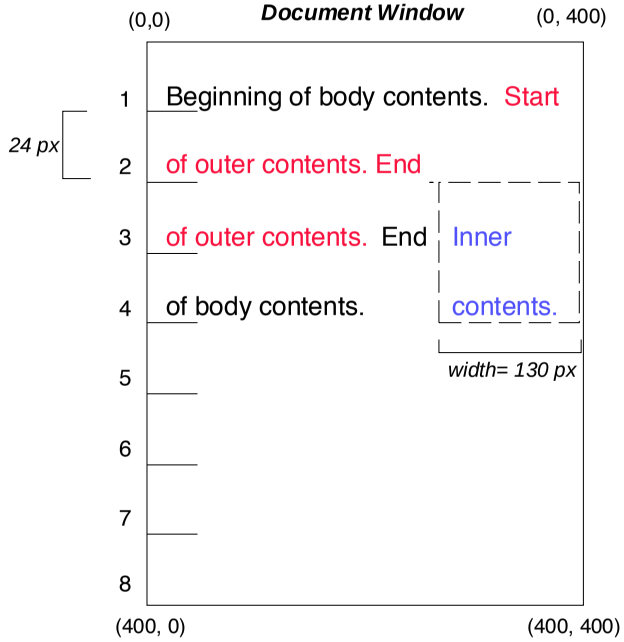
Note also that had the offset of *outer* been '-24px', the text of *outer* and the body text would have overlapped.

9.8.3 Floating a box

Now consider the effect of floating [p. 121] the *inner* element's text to the right by means of the following rules:

```
#outer { color: red }
#inner { float: right; width: 130px; color: blue }
```

Text flows normally up to the *inner* box, which is pulled out of the flow and floated to the right margin (its 'width' has been assigned explicitly). Line boxes to the left of the float are shortened, and the document's remaining text flows into them.



To show the effect of the 'clear' property, we add a *sibling* element to the example:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Comparison of positioning schemes II</TITLE>
  </HEAD>
  <BODY>
    <P>Beginning of body contents.
      <SPAN id=outer> Start of outer contents.
      <SPAN id=inner> Inner contents.</SPAN>
      <SPAN id=sibling> Sibling contents.</SPAN>
      End of outer contents.</SPAN>
```

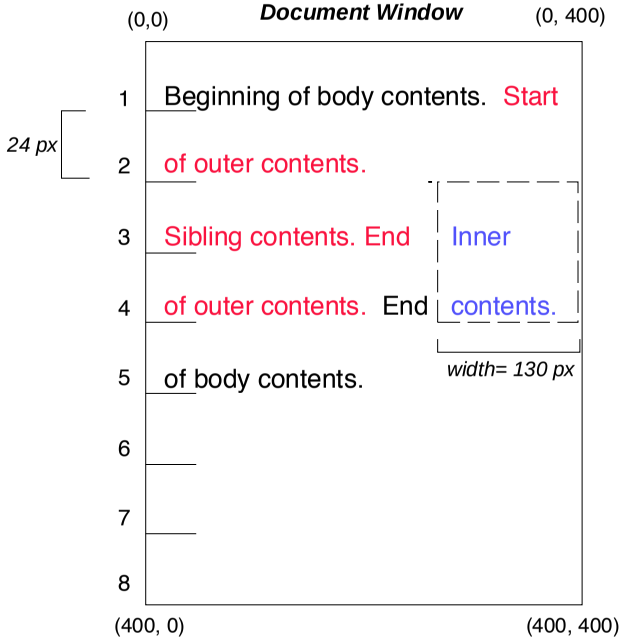


```
      End of body contents.
    </P>
  </BODY>
</HTML>
```

The following rules:

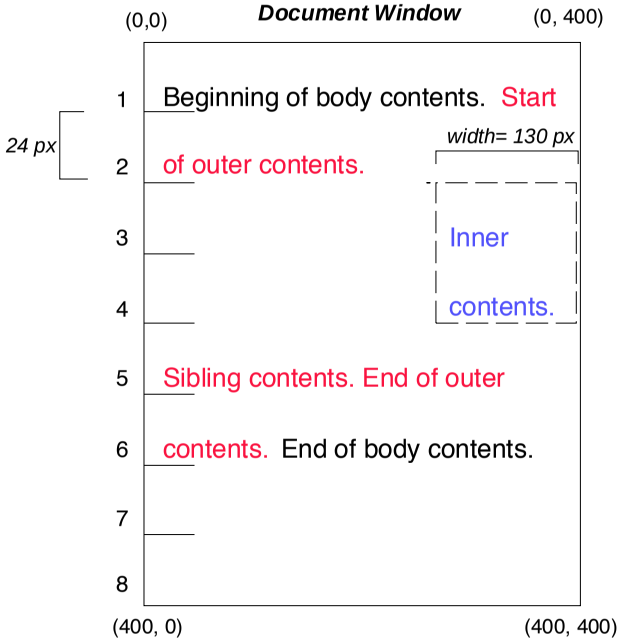
```
#inner { float: right; width: 130px; color: blue }
#sibling { color: red }
```

cause the *inner* box to float to the right as before and the document's remaining text to flow into the vacated space:



However, if the 'clear' property on the *sibling* element is set to 'right' (i.e., the generated *sibling* box will not accept a position next to floating boxes to its right), the *sibling* content begins to flow below the float:

```
#inner { float: right; width: 130px; color: blue }
#sibling { clear: right; color: red }
```

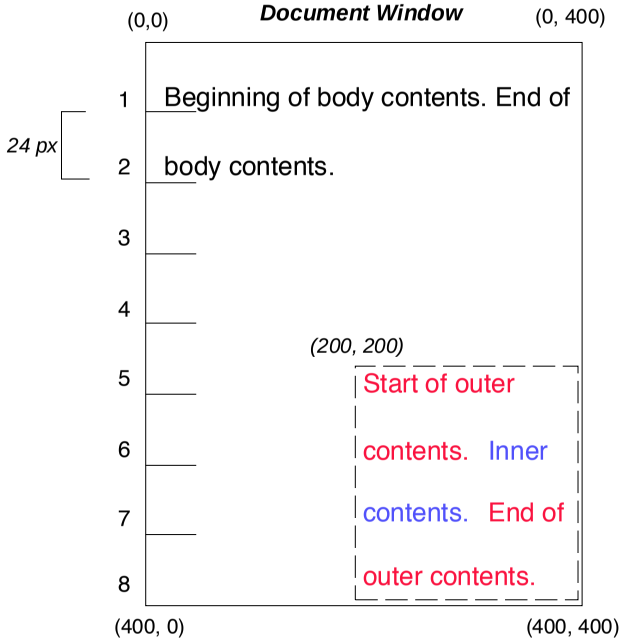


9.8.4 Absolute positioning

Finally, we consider the effect of absolute positioning [p. 129] . Consider the following CSS declarations for *outer* and *inner*:

```
#outer {
    position: absolute;
    top: 200px; left: 200px;
    width: 200px;
    color: red;
}
#inner { color: blue }
```

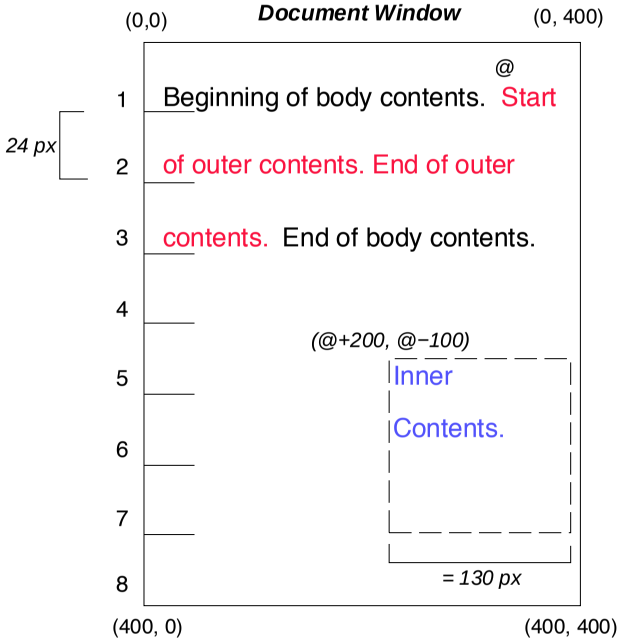
which cause the top of the *outer* box to be positioned with respect to its containing block. The containing block for a positioned box is established by the nearest positioned ancestor (or, if none exists, the initial containing block [p. 149] , as in our example). The top side of the *outer* box is '200px' below the top of the containing block and the left side is '200px' from the left side. The child box of *outer* is flowed normally with respect to its parent.



The following example shows an absolutely positioned box that is a child of a relatively positioned box. Although the parent *outer* box is not actually offset, setting its 'position' property to 'relative' means that its box may serve as the containing block for positioned descendants. Since the *outer* box is an inline box that is split across several lines, the first inline box's top and left edges (depicted by thick dashed lines in the illustration below) serve as references for 'top' and 'left' offsets.

```
#outer {
  position: relative;
  color: red
}
#inner {
  position: absolute;
  top: 200px; left: -100px;
  height: 130px; width: 130px;
  color: blue;
}
```

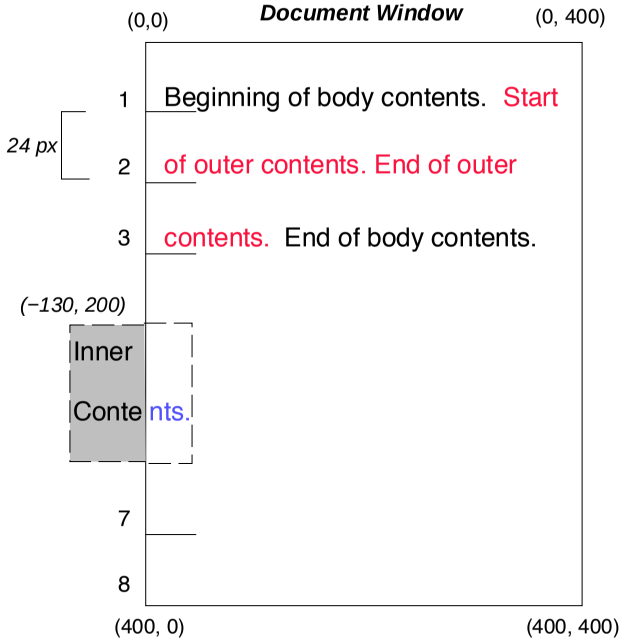
This results in something like the following:



If we do not position the *outer* box:

```
#outer { color: red }
#inner {
  position: absolute;
  top: 200px; left: -100px;
  height: 130px; width: 130px;
  color: blue;
}
```

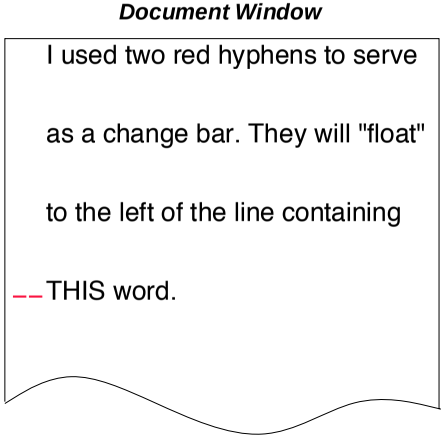
the containing block for *inner* becomes the initial containing block [p. 149] (in our example). The following illustration shows where the *inner* box would end up in this case.



Relative and absolute positioning may be used to implement change bars, as shown in the following example. The following fragment:

```
<P style="position: relative; margin-right: 10px; left: 10px;">
I used two red hyphens to serve as a change bar. They
will "float" to the left of the line containing THIS
<SPAN style="position: absolute; top: auto; left: -1em; color: red;">--</SPAN>
word.</P>
```

might result in something like:



First, the paragraph (whose containing block sides are shown in the illustration) is flowed normally. Then it is offset ’10px’ from the left edge of the containing block (thus, a right margin of ’10px’ has been reserved in anticipation of the offset). The two hyphens acting as change bars are taken out of the flow and positioned at the current line (due to ’top: auto’), ’-1em’ from the left edge of its containing block (established by the P in its final position). The result is that the change bars seem to "float" to the left of the current line.

9.9 Layered presentation

9.9.1 Specifying the stack level: the ’z-index’ property

’z-index’

Value:	auto | <integer> | inherit
Initial:	auto
Applies to:	positioned elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

For a positioned box, the ’z-index’ property specifies:

1. The stack level of the box in the current stacking context.
2. Whether the box establishes a local stacking context.

Values have the following meanings:

<integer>
This integer is the stack level of the generated box in the current stacking context. The box also establishes a local stacking context in which its stack level is ’0’.

auto
The stack level of the generated box in the current stacking context is the same as its parent’s box. The box does not establish a new local stacking context.

In this section, the expression "in front of" means closer to the user as the user faces the screen.

In CSS 2.1, each box has a position in three dimensions. In addition to their horizontal and vertical positions, boxes lie along a "z-axis" and are formatted one on top of the other. Z-axis positions are particularly relevant when boxes overlap visually. This section discusses how boxes may be positioned along the z-axis.

The order in which the rendering tree is painted onto the canvas is described in terms of stacking contexts. Stacking contexts can contain further stacking contexts. A stacking context is atomic from the point of view of its parent stacking context; boxes in other stacking contexts may not come between any of its boxes.

Each box belongs to one *stacking context*. Each box in a given stacking context has an integer *stack level*, which is its position on the z-axis relative to other boxes in the same stacking context. Boxes with greater stack levels are always formatted in front of boxes with lower stack levels. Boxes may have negative stack levels. Boxes with the same stack level in a stacking context are stacked bottom-to-top according to document tree order.

The root element forms the root stacking context. Other stacking contexts are generated by any positioned element (including relatively positioned elements) having a computed value of ’z-index’ other than ’auto’. Stacking contexts are not necessarily related to containing blocks. In future levels of CSS, other properties may introduce stacking contexts, for example ’opacity [p. ??] ’.

Each stacking context consists of the following stacking levels (from back to front):

1. the background and borders of the element forming the stacking context.
2. the stacking contexts of descendants with negative stack levels.
3. a stacking level containing in-flow non-inline-level descendants.
4. a stacking level for floats and their contents.
5. a stacking level for in-flow inline-level descendants.
6. a stacking level for positioned descendants with ’z-index: auto’, and any descendant stacking contexts with ’z-index: 0’.
7. the stacking contexts of descendants with positive stack levels.

The contents of inline blocks and inline tables are stacked as if they generated new stacking contexts, except that any elements that actually create new stacking contexts take part in the parent stacking context. They are then painted atomically in the inline stacking level.

In the following example, the stack levels of the boxes (named with their "id" attributes) are: "text2"=0, "image"=1, "text3"=2, and "text1"=3. The "text2" stack level is inherited from the root box. The others are specified with the 'z-index' property.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Z-order positioning</TITLE>
    <STYLE type="text/css">
      .pile {
        position: absolute;
        left: 2in;
        top: 2in;
        width: 3in;
        height: 3in;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <P>
      <IMG id="image" class="pile"
           src="butterfly.png" alt="A butterfly image"
           style="z-index: 1">

    <DIV id="text1" class="pile"
         style="z-index: 3">
      This text will overlay the butterfly image.
    </DIV>

    <DIV id="text2">
      This text will be beneath everything.
    </DIV>

    <DIV id="text3" class="pile"
         style="z-index: 2">
      This text will underlay text1, but overlay the butterfly image
    </DIV>
  </BODY>
</HTML>
```

This example demonstrates the notion of *transparency*. The default behavior of a box is to allow boxes behind it to be visible through transparent areas in its content. In the example, each box transparently overlays the boxes below it. This behavior can be overridden by using one of the existing background properties [p. 206] .

9.10 Text direction: the 'direction' and 'unicode-bidi' properties

Conforming [p. 34] user agents that do not support bidirectional text may ignore the 'direction' and 'unicode-bidi' properties described in this section.

The characters in certain scripts are written from right to left. In some documents, in particular those written with the Arabic or Hebrew script, and in some mixed-language contexts, text in a single (visually displayed) block may appear with mixed directionality. This phenomenon is called *bidirectionality*, or "bidi" for short.

The Unicode standard ([UNICODE], section 3.11) defines a complex algorithm for determining the proper directionality of text. The algorithm consists of an implicit part based on character properties, as well as explicit controls for embeddings and overrides. CSS 2.1 relies on this algorithm to achieve proper bidirectional rendering. The 'direction' and 'unicode-bidi' properties allow authors to specify how the elements and attributes of a document language map to this algorithm.

If a document contains right-to-left characters, and if the user agent displays these characters in right-to-left order, the user agent must apply the bidirectional algorithm. (UAs that render right-to-left characters simply because a font on the system contains them but do not support the concept of right-to-left text direction are exempt from this requirement.) This seemingly one-sided requirement reflects the fact that, although not every Hebrew or Arabic document contains mixed-directionality text, such documents are much more likely to contain left-to-right text (e.g., numbers, text from other languages) than are documents written in left-to-right languages.

Because the directionality of a text depends on the structure and semantics of the document language, these properties should in most cases be used only by designers of document type descriptions (DTDs), or authors of special documents. If a default style sheet specifies these properties, authors and users should not specify rules to override them.

The HTML 4.0 specification ([HTML40], section 8.2) defines bidirectionality behavior for HTML elements. The style sheet rules that would achieve the bidi behavior specified in [HTML40] are given in the sample style sheet [p. 294] . The HTML 4.0 specification also contains more information on bidirectionality issues.

'direction'

Value:	ltr | rtl | inherit
Initial:	ltr
Applies to:	all elements, but see prose
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies the base writing direction of blocks and the direction of embeddings and overrides (see 'unicode-bidi') for the Unicode bidirectional algorithm. In addition, it specifies the direction of table [p. 235] column layout, the direction of horizontal overflow [p. 169] , and the position of an incomplete last line in a block in case of 'text-align: justify'.

Values for this property have the following meanings:

ltr
Left-to-right direction.

rtl
Right-to-left direction.

For the 'direction' property to have any effect on inline-level elements, the 'unicode-bidi' property's value must be 'embed' or 'override'.

Note. *The 'direction' property, when specified for table column elements, is not inherited by cells in the column since columns are not the ancestors of the cells in the document tree. Thus, CSS cannot easily capture the "dir" attribute inheritance rules described in [HTML40], section 11.3.2.1.*

'unicode-bidi'

Value:	normal I embed I bidi-override I inherit
Initial:	normal
Applies to:	all elements, but see prose
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

Values for this property have the following meanings:

normal
The element does not open an additional level of embedding with respect to the bidirectional algorithm. For inline-level elements, implicit reordering works across element boundaries.

embed
If the element is inline-level, this value opens an additional level of embedding with respect to the bidirectional algorithm. The direction of this embedding level is given by the 'direction' property. Inside the element, reordering is done implicitly. This corresponds to adding a LRE (U+202A; for 'direction: ltr') or RLE (U+202B; for 'direction: rtl') at the start of the element and a PDF (U+202C) at the end of the element.

bidi-override
For inline-level elements this creates an override. For block-level elements this creates an override for inline-level descendents not within another block. This means that inside the element, reordering is strictly in sequence according to

the 'direction' property; the implicit part of the bidirectional algorithm is ignored. This corresponds to adding a LRO (U+202D; for 'direction: ltr') or RLO (U+202E; for 'direction: rtl') at the start of the element and a PDF (U+202C) at the end of the element.

The final order of characters in each block-level element is the same as if the bidi control codes had been added as described above, markup had been stripped, and the resulting character sequence had been passed to an implementation of the Unicode bidirectional algorithm for plain text that produced the same line-breaks as the styled text. In this process, non-textual entities such as images are treated as neutral characters, unless their 'unicode-bidi' property has a value other than 'normal', in which case they are treated as strong characters in the 'direction' specified for the element.

Please note that in order to be able to flow inline boxes in a uniform direction (either entirely left-to-right or entirely right-to-left), more inline boxes (including anonymous inline boxes) may have to be created, and some inline boxes may have to be split up and reordered before flowing.

Because the Unicode algorithm has a limit of *61 levels* of embedding, care should be taken not to use 'unicode-bidi' with a value other than 'normal' unless appropriate. In particular, a value of 'inherit' should be used with extreme caution. However, for elements that are, in general, intended to be displayed as blocks, a setting of 'unicode-bidi: embed' is preferred to keep the element together in case display is changed to inline (see example below).

The following example shows an XML document with bidirectional text. It illustrates an important design principle: DTD designers should take bidi into account both in the language proper (elements and attributes) and in any accompanying style sheets. The style sheets should be designed so that bidi rules are separate from other style rules. The bidi rules should not be overridden by other style sheets so that the document language's or DTD's bidi behavior is preserved.

Example(s):

In this example, lowercase letters stand for inherently left-to-right characters and uppercase letters represent inherently right-to-left characters:

```
<HEBREW>
  <PAR>HEBREW1 HEBREW2 english3 HEBREW4 HEBREW5</PAR>
  <PAR>HEBREW6 <EMPH>HEBREW7</EMPH> HEBREW8</PAR>
</HEBREW>
<ENGLISH>
  <PAR>english9 english10 english11 HEBREW12 HEBREW13</PAR>
  <PAR>english14 english15 english16</PAR>
  <PAR>english17 <HE-QUO>HEBREW18 english19 HEBREW20</HE-QUO></PAR>
</ENGLISH>
```

Since this is XML, the style sheet is responsible for setting the writing direction. This is the style sheet:

```
/* Rules for bidi */
HEBREW, HE-QUO  {direction: rtl; unicode-bidi: embed}
ENGLISH         {direction: ltr; unicode-bidi: embed}

/* Rules for presentation */
HEBREW, ENGLISH, PAR  {display: block}
EMPH                  {font-weight: bold}
```

The HEBREW element is a block with a right-to-left base direction, the ENGLISH element is a block with a left-to-right base direction. The PARs are blocks that inherit the base direction from their parents. Thus, the first two PARs are read starting at the top right, the final three are read starting at the top left. Please note that HEBREW and ENGLISH are chosen as element names for explicitness only; in general, element names should convey structure without reference to language.

The EMPH element is inline-level, and since its value for 'unicode-bidi' is 'normal' (the initial value), it has no effect on the ordering of the text. The HE-QUO element, on the other hand, creates an embedding.

The formatting of this text might look like this if the line length is long:

```
              5WERBEH 4WERBEH english3 2WERBEH 1WERBEH

                               8WERBEH 7WERBEH 6WERBEH

english9 english10 english11 13WERBEH 12WERBEH

english14 english15 english16

english17 20WERBEH english19 18WERBEH
```

Note that the HE-QUO embedding causes HEBREW18 to be to the right of english19.

If lines have to be broken, it might be more like this:

```
       2WERBEH 1WERBEH
  -EH 4WERBEH english3
                 5WERB

   -EH 7WERBEH 6WERBEH
                 8WERB

english9 english10 en-
glish11 12WERBEH
13WERBEH

english14 english15
english16

english17 18WERBEH
20WERBEH english19
```


Because HEBREW18 must be read before english19, it is on the line above english19. Just breaking the long line from the earlier formatting would not have worked. Note also that the first syllable from english19 might have fit on the previous line, but hyphenation of left-to-right words in a right-to-left context, and vice versa, is usually suppressed to avoid having to display a hyphen in the middle of a line.

---

10 Visual formatting model details

Contents

10.1 Definition of "containing block"

The position and size of an element's box(es) are sometimes calculated relative to a certain rectangle, called the *containing block* of the element. The containing block of an element is defined as follows:

1. The containing block in which the root element [p. 33] lives is chosen by the user agent. (It could be related to the viewport [p. 108] .) This containing block is called the *initial containing block*.
2. For other elements, if the element's position is 'relative' or 'static', the containing block is formed by the content edge of the nearest block-level [p. 109] , table cell or inline-block ancestor box.
3. If the element has 'position: fixed', the containing block is established by the viewport [p. 108] .
4. If the element has 'position: absolute', the containing block is established by the nearest ancestor with a 'position' of 'absolute', 'relative' or 'fixed', in the following way:
   1. In the case that the ancestor is block-level [p. 109] , the containing block is formed by the padding edge [p. 92] of the ancestor.
   2. In the case that the ancestor is inline-level, the containing block depends on the 'direction' property of the ancestor:
      1. If the 'direction' is 'ltr', the top and left of the containing block are the top and left content edges of the first box generated by the ancestor, and the bottom and right are the bottom and right content edges of the last box of the ancestor.
      2. If the 'direction' is 'rtl', the top and right are the top and right edges of the first box generated by the ancestor, and the bottom and left are the bottom and left content edges of the last box of the ancestor.

   If there is no such ancestor, the containing block is the initial containing block.

In paged media, an absolutely positioned element is positioned relative to its containing block ignoring any page breaks (as if the document were continuous). The element may subsequently be broken over several pages.

Note that a block-level element that is split over several pages may have a different width on each page and that there may be device-specific limits.

Example(s):

With no positioning, the containing blocks (C.B.) in the following document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
   <HEAD>
      <TITLE>Illustration of containing blocks</TITLE>
   </HEAD>
   <BODY id="body">
      <DIV id="div1">
      <P id="p1">This is text in the first paragraph...</P>
      <P id="p2">This is text <EM id="em1"> in the
      <STRONG id="strong1">second</STRONG> paragraph.</EM></P>
      </DIV>
   </BODY>
</HTML>
```

are established as follows:

For box generated by	C.B. is established by
html	initial C.B. (UA-dependent)
body	html
div1	body
p1	div1
p2	div1
em1	p2
strong1	p2

If we position "div1":

```
#div1 { position: absolute; left: 50px; top: 50px }
```

its containing block is no longer "body"; it becomes the initial containing block (since there are no other positioned ancestor boxes).

If we position "em1" as well:

```
#div1 { position: absolute; left: 50px; top: 50px }
#em1  { position: absolute; left: 100px; top: 100px }
```

the table of containing blocks becomes:

For box generated by	C.B. is established by
html	initial C.B. (UA-dependent)
body	html
div1	initial C.B.
p1	div1
p2	div1
em1	div1
strong1	em1

By positioning "em1", its containing block becomes the nearest positioned ancestor box (i.e., that generated by "div1").

10.2 Content width: the 'width' property

'width'

Value:	<length> I <percentage> I auto I inherit
Initial:	auto
Applies to:	all elements but non-replaced inline elements, table rows, and row groups
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length; 'auto' if the property does not apply

This property specifies the content width [p. 92] of boxes generated by block-level and replaced [p. 32] elements.

This property does not apply to non-replaced inline-level [p. 111] elements. The content width of a non-replaced inline element's boxes is that of the rendered content within them (*before* any relative offset of children). Recall that inline boxes flow into line boxes [p. 118] . The width of line boxes is given by the their containing block [p. 108] , but may be shorted by the presence of floats [p. 121] .

The width of a replaced element's box is intrinsic [p. 32] and may be scaled by the user agent if the value of this property is different than 'auto'.

Values have the following meanings:

<length>
Specifies the width of the content area using a length unit.

<percentage>
Specifies a percentage width. The percentage is calculated with respect to the width of the generated box's containing block [p. 108] . If the containing block's width depends on this element's width, then the resulting layout is undefined in CSS 2.1. Note: For absolutely positioned elements whose containing block is based on a block-level element, the percentage is calculated with respect to the width of the *padding box* of that element. This is a change from CSS1, where the percentage width was always calculated with respect to the *content box* of the parent element.

auto
The width depends on the values of other properties. See the sections below.

Negative values for 'width' are illegal.

Example(s):

For example, the following rule fixes the content width of paragraphs at 100 pixels:

```
p { width: 100px }
```

10.3 Calculating widths and margins

The values of an element's 'width', 'margin-left', 'margin-right', 'left' and 'right' properties as used for layout depend on the type of box generated and on each other. (The value used for layout is sometimes referred to as the *used value*.) In principle, the values used are the same as the computed values, with 'auto' replaced by some suitable value, and percentages calculated based on the containing block, but there are exceptions. The following situations need to be distinguished:

1. inline, non-replaced elements
2. inline, replaced elements
3. block-level, non-replaced elements in normal flow
4. block-level, replaced elements in normal flow
5. floating, non-replaced elements
6. floating, replaced elements
7. absolutely positioned, non-replaced elements
8. absolutely positioned, replaced elements
9. 'inline-block', non-replaced elements in normal flow
10. 'inline-block', replaced elements in normal flow

For Points 1-6 and 9-10, the values of 'left' and 'right' used for layout are determined by the rules in section 9.4.3. [p. 120]

10.3.1 Inline, non-replaced elements

The 'width' property does not apply. A computed value of 'auto' for 'left', 'right', 'margin-left' or 'margin-right' becomes a used value of '0'.

10.3.2 Inline, replaced elements

A computed value of 'auto' for 'margin-left' or 'margin-right' becomes a used value of '0'. If 'width' has a computed value of 'auto' and 'height' also has a computed value of 'auto', the element's intrinsic [p. 32] width is the used value of 'width'. If 'width' has a computed value of 'auto' and 'height' has some other computed value, then the used value of 'width' is:

```
(intrinsic width) * ( (used height) / (intrinsic height) )
```

10.3.3 Block-level, non-replaced elements in normal flow

The following constraints must hold between the used values of the other properties:

'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' + 'border-right-width' + 'margin-right' = width of containing block [p. 149]

If all of the above have a computed value other than 'auto', the values are said to be "over-constrained" and one of the used values will have to be different from its computed value. If the 'direction' property has the value 'ltr', the specified value of 'margin-right' is ignored and the value is calculated so as to make the equality true. If the value of 'direction' is 'rtl', this happens to 'margin-left' instead.

If there is exactly one value specified as 'auto', its used value follows from the equality.

If 'width' is set to 'auto', any other 'auto' values become '0' and 'width' follows from the resulting equality.

If both 'margin-left' and 'margin-right' are 'auto', their used values are equal. This horizontally centers the element with respect to the edges of the containing block.

10.3.4 Block-level, replaced elements in normal flow

The used value of 'width' is determined as for inline replaced elements [p. 153] . If one of the margins is 'auto', its used value is given by the constraints [p. 153] above. Furthermore, if both margins are 'auto', their used values are equal.

10.3.5 Floating, non-replaced elements

If 'margin-left', or 'margin-right' are computed as 'auto', their used value is '0'.

If 'width' is computed as 'auto', the used value is the "shrink-to-fit" width.

Calculation of the shrink-to-fit width is similar to calculating the width of a table cell using the automatic table layout algorithm. Roughly: calculate the preferred width by formatting the content without breaking lines other than where explicit line breaks occur, and also calculate the preferred *minimum* width, e.g., by trying all possible line breaks. CSS 2.1 does not define the exact algorithm. Thirdly, find the *available width*: in this case, this is the width of the containing block minus 'margin-left' and 'margin-right'.

Then the shrink-to-fit width is: min(max(preferred minimum width, available width), preferred width).

10.3.6 Floating, replaced elements

If 'margin-left' or 'margin-right' are computed as 'auto', their used value is '0'. The used value of 'width' is determined as for inline replaced elements [p. 153] .

10.3.7 Absolutely positioned, non-replaced elements

For the purposes of this section and the next, the term "static position" (of an element) refers, roughly, to the position an element would have had in the normal flow. More precisely:

- The static position for 'left' is the distance from the left edge of the containing block to the left margin edge of a hypothetical box that would have been the first box of the element if its 'position' property had been 'static'. The value is negative if the hypothetical box is to the left of the containing block.
- The static position for 'right' is the distance from the right edge of the containing block to the right margin edge of the same hypothetical box as above. The value is positive if the hypothetical box is to the left of the containing block's edge.

But rather than actually calculating the dimensions of that hypothetical box, user agents are free to make a guess at its probable position.

For the purposes of calculating the static position, the containing block of fixed positioned elements is the initial containing block instead of the viewport.

The constraint that determines the used values for these elements is:

'left' + 'margin-left' + 'border-left-width' + 'padding-left' + 'width' + 'padding-right' + 'border-right-width' + 'margin-right' + 'right' = width of containing block

If all three of 'left', 'width', and 'right' are 'auto': First set any 'auto' values for 'margin-left' and 'margin-right' to 0. Then, if 'direction' is 'ltr' set 'left' to the static position [p. 154] and apply rule number three below; otherwise, set 'right' to the static position [p. 154] and apply rule number one below.

If none of the three is 'auto': If both 'margin-left' and 'margin-right' are 'auto', solve the equation under the extra constraint that the two margins get equal values. If one of 'margin-left' or 'margin-right' is 'auto', solve the equation for that value. If the values are over-constrained, ignore the value for 'left' (in case 'direction' is 'rtl') or 'right' (in case 'direction' is 'ltr') and solve for that value.

Otherwise, set 'auto' values for 'margin-left' and 'margin-right' to 0, and pick the one of the following six rules that applies.

1. 'left' and 'width' are 'auto' and 'right' is not 'auto', then the width is shrink-to-fit. Then solve for 'left'
2. 'left' and 'right' are 'auto' and 'width' is not 'auto', then if 'direction' is 'ltr' set 'left' to the static position [p. 154] , otherwise set 'right' to the static position [p. 154] . Then solve for 'left' (if 'direction is 'rtl') or 'right' (if 'direction' is 'ltr').
3. 'width' and 'right' are 'auto' and 'left' is not 'auto', then the width is shrink-to-fit . Then solve for 'right'
4. 'left' is 'auto', 'width' and 'right' are not 'auto', then solve for 'left'
5. 'width' is 'auto', 'left' and 'right' are not 'auto', then solve for 'width'
6. 'right' is 'auto', 'left' and 'width' are not 'auto', then solve for 'right'

Calculation of the shrink-to-fit width is similar to calculating the width of a table cell using the automatic table layout algorithm. Roughly: calculate the preferred width by formatting the content without breaking lines other than where explicit line breaks occur, and also calculate the preferred *minimum* width, e.g., by trying all possible

line breaks. CSS 2.1 does not define the exact algorithm. Thirdly, calculate the *available width*: this is found by solving for 'width' after setting 'left' (in case 1) or 'right' (in case 3) to 0.

Then the shrink-to-fit width is: min(max(preferred minimum width, available width), preferred width).

10.3.8 Absolutely positioned, replaced elements

This situation is similar to the previous one, except that the element has an intrinsic [p. 32] width. The sequence of substitutions is now:

1. The used value of 'width' is determined as for inline replaced elements [p. 153] .
2. If 'left' has the value 'auto' while 'direction' is 'ltr', replace 'auto' with the static position [p. 154] .
3. If 'right' has the value 'auto' while 'direction' is 'rtl', replace 'auto' with the static position [p. 154] .
4. If 'left' or 'right' are 'auto', replace any 'auto' on 'margin-left' or 'margin-right' with '0'.
5. If at this point both 'margin-left' and 'margin-right' are still 'auto', solve the equation under the extra constraint that the two margins must get equal values.
6. If at this point there is only one 'auto' left, solve the equation for that value.
7. If at this point the values are over-constrained, ignore the value for either 'left' (in case 'direction' is 'rtl') or 'right' (in case 'direction' is 'ltr') and solve for that value.

10.3.9 'Inline-block', non-replaced elements in normal flow

If 'width' is 'auto', the used value is the shrink-to-fit [p. 154] width as for floating elements.

A computed value of 'auto' for 'margin-left' or 'margin-right' becomes a used value of '0'.

10.3.10 'Inline-block', replaced elements in normal flow

Exactly as inline replaced elements. [p. 153]

10.4 Minimum and maximum widths: 'min-width' and 'max-width'

'min-width'

Value:	<length>	<percentage>	inherit
Initial:	0
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length

'max-width'

Value:	<length>	<percentage>	none	inherit
Initial:	none
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length or 'none'

These two properties allow authors to constrain box widths to a certain range. Values have the following meanings:

<length>
Specifies a fixed minimum or maximum used width.
<percentage>
Specifies a percentage for determining the used value. The percentage is calculated with respect to the width of the generated box's containing block [p. 108] .
none
(Only on 'max-width') No limit on the width of the box.

Negative values for 'min-width' and 'max-width' are illegal.

The following algorithm describes how the two properties influence the used value [p. 80] of the 'width' property:

1. The tentative used width is calculated (without 'min-width' and 'max-width') following the rules under "Calculating widths and margins" [p. 153] above.
2. If the tentative used width is greater than 'max-width', the rules above [p. 153] are applied again, but this time using the computed value of 'max-width' as the computed value for 'width'.
3. If the resulting width is smaller than 'min-width', the rules above [p. 153] are applied again, but this time using the value of 'min-width' as the computed value for 'width'.

However, for replaced elements with both 'width' and 'height' specified as 'auto', the algorithm is as follows:

1. Select from the following list of width-height pairs (a, b) the first one that satisfies the two constraints min-width ≤ a ≤ max(min-width, max-width) and min-height ≤ b ≤ max(min-height, max-height). The resulting pair gives the used width and height for the element. In this list, Wi and Hi stand for the intrinsic width and height, respectively.
   1. (Wi, Hi)
   2. (max(Wi, min-width), max(Wi, min-width)*Hi/Wi)
   3. (max(Hi, min-height)*Wi/Hi, max(Hi, min-height))
   4. (min(Wi, max-width), min(Wi, max-width)*Hi/Wi)
   5. (min(Hi, max-height)*Wi/Hi, min(Hi, max-height))
   6. (max(Wi, min-width), min(Hi, max-height))
   7. (min(Wi, max-width), max(Hi, min-height))
   8. (max(Wi, min-width), max(Hi, min-height))
   9. (min(Wi, max-width), min(Hi, max-height))
2. Then apply the rules under "Calculating widths and margins" [p. 153] above, as if 'width' were computed as this value.

10.5 Content height: the 'height' property

'height'

Value:	<length>	<percentage>	auto	inherit
Initial:	auto
Applies to:	all elements but non-replaced inline elements, table columns, and column groups
Inherited:	no
Percentages:	see prose
Media:	visual
Computed value:	the percentage as specified or the absolute length; 'auto' if the property does not apply

This property specifies the content height [p. 92] of boxes generated by block-level, inline-block and replaced [p. 32] elements.

This property does not apply to non-replaced inline-level [p. 111] elements. See the section on computing heights and margins for non-replaced inline elements [p. 160] for the rules used instead.

Values have the following meanings:

<length>
Specifies the height of the content area using a length value.

<percentage>
Specifies a percentage height. The percentage is calculated with respect to the height of the generated box's containing block [p. 108] . If the height of the containing block is not specified explicitly (i.e., it depends on content height), and this element is not absolutely positioned, the value is interpreted like 'auto'. A percentage height on the root element [p. 33] is relative to the viewport [p. 108] .

auto
The height depends on the values of other properties. See the prose below.

Note that the height of the containing block of an absolutely positioned element is independent of the size of the element itself, and thus a percentage height on such an element can always be resolved. However, it may be that the height is not known until elements that come later in the document have been processed.

Negative values for 'height' are illegal.

Example(s):

For example, the following rule sets the content height of paragraphs to 100 pixels:

```
p { height: 100px }
```

Paragraphs of which the height of the contents exceeds 100 pixels will overflow [p. 169] according to the 'overflow' property.

10.6 Calculating heights and margins

For calculating the values of 'top', 'margin-top', 'height', 'margin-bottom', and 'bottom' a distinction must be made between various kinds of boxes:

1. inline, non-replaced elements
2. inline, replaced elements
3. block-level, non-replaced elements in normal flow
4. block-level, replaced elements in normal flow
5. floating, non-replaced elements
6. floating, replaced elements
7. absolutely positioned, non-replaced elements
8. absolutely positioned, replaced elements
9. 'inline-block', non-replaced elements in normal flow
10. 'inline-block', replaced elements in normal flow

For Points 1-6 and 9-10, the used values of 'top' and 'bottom' are determined by the rules in section 9.4.3.

10.6.1 Inline, non-replaced elements

The 'height' property doesn't apply. The height of the content area should be based on the font, but this specification does not specify how. A UA may, e.g., use the em-box or the maximum ascender and descender of the font. (The latter would ensure that glyphs with parts above or below the em-box still fall within the content area, but leads to differently sized boxes for different fonts; the formed would ensure authors can control background styling relative to the 'line-height', but leads to glyphs painting outside their content area.)

Note: level 3 of CSS will probably include a property to select which measure of the font is used for the content height.

The vertical padding, border and margin of an inline, non-replaced box start at the top and bottom of the content area, not the 'line-height'. But only the 'line-height' is used to calculate the height of the line box.

If more than one font is used (this could happen when glyphs are found in different fonts), the height of the content area is not defined by this specification. However, we suggest that the height is chosen such that the content area is just high enough for either (1) the em-boxes, or (2) the maximum ascenders and descenders, of *all* the fonts in the element. Note that this may be larger than any of the font sizes involved, depending on the baseline alignment of the fonts.

10.6.2 Inline replaced elements, block-level replaced elements in normal flow, 'inline-block' replaced elements in normal flow and floating replaced elements

If 'margin-top', or 'margin-bottom' are 'auto', their used value is 0. If 'height' has a computed value of 'auto' and 'width' also has a computed value of 'auto', the element's intrinsic height is the used value of 'height'. If 'height' has a computed value of 'auto' and 'width' has some other computed value, then the used value of 'height' is:

```
(intrinsic height) * ( (used width) / (intrinsic width) )
```

10.6.3 Block-level and 'inline-block', non-replaced elements in normal flow

If 'margin-top', or 'margin-bottom' are 'auto', their used value is 0. If 'height' is 'auto', the height depends on whether the element has any block-level children and whether it has padding or borders:

If it only has inline-level children, the height is the distance between the top of the topmost line box and the bottom of the bottommost line box.

If it has block-level children, the height is the distance between the top border-edge of the topmost block-level child box that doesn't have margins collapsed through it [p. 97] and the bottom border-edge of the bottommost block-level child box that

doesn't have margins collapsed through it. However, if the element has a non-zero top padding and/or top border, then the content starts at the top *margin* edge of the topmost child. (The first case expresses the fact that the top and bottom margins of the element collapse [p. 97] with those of the topmost and bottommost children, while in the second case the presence of the padding/border prevents the top margins from collapsing [p. 97] .) Similarly, if the element has a non-zero bottom padding and/or bottom border, then the content ends at the bottom *margin* edge of the bottommost child.

Only children in the normal flow are taken into account (i.e., floating boxes and absolutely positioned boxes are ignored, and relatively positioned boxes are considered without their offset). Note that the child box may be an anonymous block box. [p. 109]

10.6.4 Absolutely positioned, non-replaced elements

For the purposes of this section and the next, the term "static position" (of an element) refers, roughly, to the position an element would have had in the normal flow. More precisely, the static position for 'top' is the distance from the top edge of the containing block to the top margin edge of a hypothetical box that would have been the first box of the element if its 'position' property had been 'static'. The value is negative if the hypothetical box is above the containing block.

But rather than actually calculating the dimensions of that hypothetical box, user agents are free to make a guess at its probable position.

For the purposes of calculating the static position, the containing block of fixed positioned elements is the initial containing block instead of the viewport.

For absolutely positioned elements, the used values of the vertical dimensions must satisfy this constraint:

'top' + 'margin-top' + 'border-top-width' + 'padding-top' + 'height' + 'padding-bottom' + 'border-bottom-width' + 'margin-bottom' + 'bottom' = height of containing block

If all three of 'top', 'height', and 'bottom' are auto, set 'top' to the static position and apply rule number three below.

If none of the three are 'auto': If both 'margin-top' and 'margin-bottom' are 'auto', solve the equation under the extra constraint that the two margins get equal values. If one of 'margin-top' or 'margin-bottom' is 'auto', solve the equation for that value. If the values are over-constrained, ignore the value for 'bottom' and solve for that value.

Otherwise, pick the one of the following six rules that applies.

1. 'top' and 'height' are 'auto' and 'bottom' is not 'auto', then the height is based on the content, set 'auto' values for 'margin-top' and 'margin-bottom' to 0, and solve for 'top'
2. 'top' and 'bottom' are 'auto' and 'height' is not 'auto', then set 'top' to the static position, set 'auto' values for 'margin-top' and 'margin-bottom' to 0, and solve for 'bottom'
3. 'height' and 'bottom' are 'auto' and 'top' is not 'auto', then the height is based on the content, set 'auto' values for 'margin-top' and 'margin-bottom' to 0, and solve for 'bottom'
4. 'top' is 'auto', 'height' and 'bottom' are not 'auto', then set 'auto' values for 'margin-top' and 'margin-bottom' to 0, and solve for 'top'
5. 'height' is 'auto', 'top' and 'bottom' are not 'auto', then 'auto' values for 'margin-top' and 'margin-bottom' are set to 0 and solve for 'height'
6. 'bottom' is 'auto', 'top' and 'height' are not 'auto', then set 'auto' values for 'margin-top' and 'margin-bottom' to 0 and solve for 'bottom'

10.6.5 Absolutely positioned, replaced elements

This situation is similar to the previous one, except that the element has an intrinsic [p. 32] height. The sequence of substitutions is now:

1. The used value of 'height' is determined as for inline replaced elements [p. 160] .
2. If 'top' has the value 'auto', replace it with the element's static position.
3. If 'bottom' is 'auto', replace any 'auto' on 'margin-top' or 'margin-bottom' with '0'.
4. If at this point both 'margin-top' and 'margin-bottom' are still 'auto', solve the equation under the extra constraint that the two margins must get equal values.
5. If at this point there is only one 'auto' left, solve the equation for that value.
6. If at this point the values are over-constrained, ignore the value for 'bottom' and solve for that value.

10.6.6 Floating, non-replaced elements

If 'margin-top', or 'margin-bottom' are 'auto', their used value is 0. If 'height' is 'auto', the height depends on the element's descendants:

If it only has inline-level children, the height is the distance between the top of the topmost line box and the bottom of the bottommost line box.

If it has block-level children, the height is the distance between the top margin-edge of the topmost block-level child box and the bottom margin-edge of the bottommost block-level child box. (Note that the margins of the float and its children do not collapse together.)

Absolutely positioned children are ignored, and relatively positioned boxes are considered without their offset. Note that the child box may be an anonymous block box. [p. 109]

In addition, if the element has any floating descendants whose top margin edge is above the top established above or whose bottom margin edge is below the bottom, then the height is increased to include those edges. Only floats that are children of the element itself or of descendants in the normal flow are taken into account, i.e., floats inside absolutely positioned descendants are not.

10.7 Minimum and maximum heights: ’min-height’ and ’max-height’

It is sometimes useful to constrain the height of elements to a certain range. Two properties offer this functionality:

’min-height’

Value:	<length> I <percentage> I inherit
Initial:	0
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	see prose
Media:	visual
Computed value:	the percentage as specified or the absolute length

’max-height’

Value:	<length> I <percentage> I none I inherit
Initial:	none
Applies to:	all elements except non-replaced inline elements and table elements
Inherited:	no
Percentages:	see prose
Media:	visual
Computed value:	the percentage as specified or the absolute length or ’none’

These two properties allow authors to constrain box heights to a certain range. Values have the following meanings:

<length>
Specifies a fixed minimum or maximum computed height.

<percentage>
Specifies a percentage for determining the used value. The percentage is calculated with respect to the height of the generated box’s containing block [p. 108] . If the height of the containing block is not specified explicitly (i.e., it depends on content height), the percentage value is treated as ’0’ (for ’min-height’) or ’none’ (for ’max-height’).

none
(Only on ’max-height’) No limit on the height of the box.

Negative values for ’min-height’ and ’max-height’ are illegal.

The following algorithm describes how the two properties influence the computed value [p. 80] of the ’height’ property:

1. The tentative used height is calculated (without ’min-height’ and ’max-height’) following the rules under "Calculating heights and margins" [p. 159] above.
2. If this tentative height is greater than ’max-height’, the rules above [p. 159] are applied again, but this time using the value of ’max-height’ as the computed value for ’height’.
3. If the resulting height is smaller than ’min-height’, the rules above [p. 159] are applied again, but this time using the value of ’min-height’ as the computed value for ’height’.

However, for replaced elements with both ’width’ and ’height’ computed as ’auto’, use the algorithm under Minimum and maximum widths [p. 156] above to find the used width and height. Then apply the rules under "Computing heights and margins" [p. 159] above, using the resulting width and height as if they were the computed values.

10.8 Line height calculations: the ’line-height’ and ’vertical-align’ properties

As described in the section on inline formatting contexts [p. 118] , user agents flow inline boxes into a vertical stack of line boxes [p. 118] . The height of a line box is determined as follows:

1. The height of each inline box in the line box is calculated (see "Calculating heights and margins" [p. 159] and the ’line-height’ property).
2. The inline boxes are aligned vertically according to their ’vertical-align’ property.
3. The line box height is the distance between the uppermost box top and the lowermost box bottom.

Empty inline elements generate empty inline boxes, but these boxes still have margins, padding, borders and a line height, and thus influence these calculations just like elements with content.

10.8.1 Leading and half-leading

Since the value of ’line-height’ may be different from the height of the content area there may be space above and below rendered glyphs. The difference between the content height and the used value of ’line-height’ is called the *leading*. Half the leading is called the *half-leading*.

User agents center glyphs vertically in an inline box, adding half-leading on the top and bottom. For example, if a piece of text is ’12px’ high and the ’line-height’ value is ’14px’, 2pxs of extra space should be added: 1px above and 1px below the letters. (This applies to empty boxes as well, as if the empty box contained an infinitesimally narrow letter.)

When the 'line-height' value is less than the content height, the final inline box height will be less than the font size and the rendered glyphs will "bleed" outside the box. If such a box touches the edge of a line box, the rendered glyphs will also "bleed" into the adjacent line box.

Although margins, borders, and padding of non-replaced elements do not enter into the line box calculation, they are still rendered around inline boxes. This means that if the height specified by 'line-height' is less than the content height of contained boxes, backgrounds and colors of padding and borders may "bleed" into adjacent line boxes. User agents should render the boxes in document order. This will cause the borders on subsequent lines to paint over the borders and text of previous lines.

'line-height'

Value:	normal | <number> | <length> | <percentage> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	refer to the font size of the element itself
Media:	visual
Computed value:	for <length> and <percentage> the absolute value; otherwise as specified

If the property is set on a block-level [p. 109] element whose content is composed of inline-level [p. 111] elements, it specifies the *minimal* height of line boxes within the element. The minimum height consist of a minimum height above the block's baseline and a minimum depth below it, exactly as if each line box starts with a zero-width inline box with the block's font and line height properties (what TEX calls a "strut").

If the property is set on an inline-level [p. 111] element, it specifies the height that is used in the calculation of the line box height (except for inline replaced [p. 32] elements, where the height of the box is given by the 'height' property).

Values for this property have the following meanings:

normal
: Tells user agents to set the used value to a "reasonable" value based on the font of the element. The value has the same meaning as <number>. We recommend a used value for 'normal' between 1.0 to 1.2. The computed value [p. 80] is 'normal'.

<length>
: The specified length is used in the calculation of the line box height. Negative values are illegal.

<number>
: The used value of the property is this number multiplied by the element's font size. Negative values are illegal. The computed value [p. 80] is the same as the specified value.

<percentage>
: The computed value [p. 80] of the property is this percentage multiplied by the element's computed font size. Negative values are illegal.

Example(s):

The three rules in the example below have the same resultant line height:

```
div { line-height: 1.2; font-size: 10pt }     /* number */
div { line-height: 1.2em; font-size: 10pt }   /* length */
div { line-height: 120%; font-size: 10pt }    /* percentage */
```

When an element contains text that is rendered in more than one font, user agents may determine the 'line-height' value according to the largest font size.

Generally, when there is only one value of 'line-height' for all inline boxes in a paragraph (and no tall images), the above will ensure that baselines of successive lines are exactly 'line-height' apart. This is important when columns of text in different fonts have to be aligned, for example in a table.

'vertical-align'

Value:	baseline | sub | super | top | text-top | middle | bottom | text-bottom | <percentage> | <length> | inherit
Initial:	baseline
Applies to:	inline-level and 'table-cell' elements
Inherited:	no
Percentages:	refer to the 'line-height' of the element itself
Media:	visual
Computed value:	for <percentage> and <length> the absolute length, otherwise as specified

This property affects the vertical positioning inside a line box of the boxes generated by an inline-level element. The following values only have meaning with respect to a parent inline-level element, or to the strut [p. 165] of a parent block-level element.

Note. *Values of this property have slightly different meanings in the context of tables. Please consult the section on table height algorithms [p. 249] for details.*

baseline
: Align the baseline of the box with the baseline of the parent box. If the box doesn't have a baseline, align the bottom margin edge with the parent's baseline.

middle
: Align the vertical midpoint of the box with the baseline of the parent box plus half the x-height of the parent.

sub
: Lower the baseline of the box to the proper position for subscripts of the parent's box. (This value has no effect on the font size of the element's text.)

super
Raise the baseline of the box to the proper position for superscripts of the parent's box. (This value has no effect on the font size of the element's text.)
text-top
Align the top of the box with the top of the parent element's font.
text-bottom
Align the bottom of the box with the bottom of the parent element's font.
<percentage>
Raise (positive value) or lower (negative value) the box by this distance (a percentage of the 'line-height' value). The value '0%' means the same as 'baseline'.
<length>
Raise (positive value) or lower (negative value) the box by this distance. The value '0cm' means the same as 'baseline'.

top
Align the top of the box with the top of the line box.
bottom
Align the bottom of the box with the bottom of the line box.

The baseline of an 'inline-table' is the baseline of the first row of the table.

A UA should use the baseline of the last line box in the normal flow in the element as the baseline of an 'inline-block', or the element's bottom margin edge, if there is none.

Visual formatting model details

11 Visual effects

Contents

11.1 Overflow and clipping

Generally, the content of a block box is confined to the content edges of the box. In certain cases, a box may *overflow*, meaning its content lies partly or entirely outside of the box, e.g.:

- A line cannot be broken, causing the line box to be wider than the block box.
- A block-level box is too wide for the containing block. This may happen when an element's 'width' property has a value that causes the generated block box to spill over sides of the containing block.
- An element's height exceeds an explicit height assigned to the containing block (i.e., the containing block's height is determined by the 'height' property, not by content height).
- A descendent box is positioned absolutely [p. 129] , partly outside the box. Such boxes are *not* clipped by the overflow property on their ancestors.
- A descendent box has negative margins [p. 95] , causing it to be positioned partly outside the box.
- The 'text-indent' property causes an inline box to hang off either the left or right edge of the block box.

Whenever overflow occurs, the 'overflow' property specifies whether a box is clipped to its content box, and if so, whether a scrolling mechanism is provided to access any clipped out content.

11.1.1 Overflow: the 'overflow' property

'overflow'

Value:	visible | hidden | scroll | auto | inherit
Initial:	visible
Applies to:	block-level and replaced elements, table cells, inline blocks
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies whether content of a block-level element is clipped when it overflows the element's box. It affects the clipping of all of the element's content except any descendant elements (and their respective content and descendants) whose containing block is the viewport or an ancestor of the element. Values have the following meanings:

visible
: This value indicates that content is not clipped, i.e., it may be rendered outside the block box.

hidden
: This value indicates that the content is clipped and that no scrolling user interface should be provided to view the content outside the clipping region; users will not have access to clipped content.

scroll
: This value indicates that the content is clipped and that if the user agent uses a scrolling mechanism that is visible on the screen (such as a scroll bar or a panner), that mechanism should be displayed for a box whether or not any of its content is clipped. This avoids any problem with scrollbars appearing and disappearing in a dynamic environment. When this value is specified and the target medium is 'print', overflowing content may be printed.

auto
: The behavior of the 'auto' value is user agent-dependent, but should cause a scrolling mechanism to be provided for overflowing boxes.

Even if 'overflow' is set to 'visible', content may be clipped to a UA's document window by the native operating environment.

HTML UAs may apply the overflow property from the BODY or HTML elements to the viewport.

In the case of a scrollbar being placed on an edge of the element's box, it should be inserted between the inner border edge and the outer padding edge.

Example(s):

Consider the following example of a block quotation (`<blockquote>`) that is too big for its containing block (established by a `<div>`). Here is the source document:

```
<div>
<blockquote>
<p>I didn't like the play, but then I saw
it under adverse conditions - the curtain was up.</p>
<cite>- Groucho Marx</cite>
</blockquote>
</div>
```

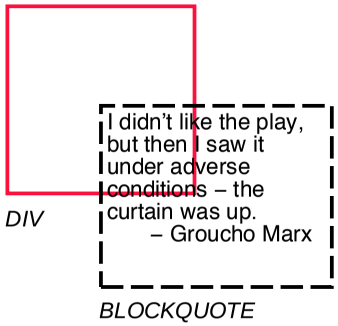
Here is the style sheet controlling the sizes and style of the generated boxes:

```
div { width : 100px; height: 100px;
      border: thin solid red;
      }
```

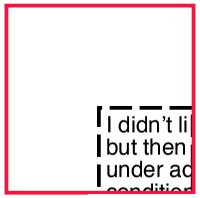
```
blockquote   { width : 125px; height : 100px;
      margin-top: 50px; margin-left: 50px;
      border: thin dashed black
      }

cite { display: block;
       text-align : right;
       border: none
       }
```

The initial value of 'overflow' is 'visible', so the <blockquote> would be formatted without clipping, something like this:



Setting 'overflow' to 'hidden' for the , on the other hand, causes the <blockquote> to be clipped by the containing block:



A value of 'scroll' would tell UAs that support a visible scrolling mechanism to display one so that users could access the clipped content.

Finally, consider this case where an absolutely positioned element is mixed with an overflow parent.

Stylesheet:

```
container { position: relative; border: solid; }
scroller { overflow: scroll; height: 5em; margin: 5em; }
satellite { position: absolute; top: 0; }
body { height: 10em; }
```


Document fragment:

```
<container>
 <scroller>
  <satellite/>
  <body/>
 </scroller>
</container>
```

In this example, the "scroller" element will not scroll the "satellite" element, because the latter's containing block is outside the element whose overflow is being clipped and scrolled.

11.1.2 Clipping: the 'clip' property

A *clipping region* defines what portion of an element's border box is visible. By default, the clipping region has the same size and shape as the element's border box. However, the clipping region may be modified by the 'clip' property.

'clip'

Value:	<shape> I auto I inherit
Initial:	auto
Applies to:	absolutely positioned elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	For rectangle values, a rectangle consisting of four computed lengths; otherwise, as specified

The 'clip' property applies only to absolutely positioned elements. Values have the following meanings:

auto
The element does not clip.

<shape>
In CSS 2.1, the only valid <shape> value is: rect(<top>, <right>, <bottom>, <left>) where <top> and <bottom> specify offsets from the top border edge of the box, and <right>, and <left> specify offsets from the left border edge of the box in left-to-right text and from the right border edge of the box in right-to-left text. Authors should separate offset values with commas. User agents must support separation with commas, but may also support separation without commas, because a previous version of this specification was ambiguous in this respect.

<top>, <right>, <bottom>, and <left> may either have a <length> value or 'auto'. Negative lengths are permitted. The value 'auto' means that a given edge of the clipping region will be the same as the edge of the element's generated border box (i.e., 'auto' means the same as '0' for <top> and <left> (in left-to-right

text, <right> in right-to-left text), the same as the computed value of the height plus the sum of vertical padding and border widths for <bottom>, and the same as the computed value of the width plus the sum of the horizontal padding and border widths for <right> (in left-to-right text, <left> in right-to-left text), such that four 'auto' values result in the clipping region being the same as the element's border box).

When coordinates are rounded to pixel coordinates, care should be taken that no pixels remain visible when <left> and <right> have the same value (or <top> and <bottom> have the same value), and conversely that no pixels within the element's border box remain hidden when these values are 'auto'.

An element's clipping region clips out any aspect of the element (e.g. content, children, background, borders, text decoration, outline and visible scrolling mechanism — if any) that is outside the clipping region.

The element's ancestors may also clip portions of their content (e.g. via their own 'clip' property and/or if their 'overflow' property is not 'visible'); what is rendered is the cumulative intersection.

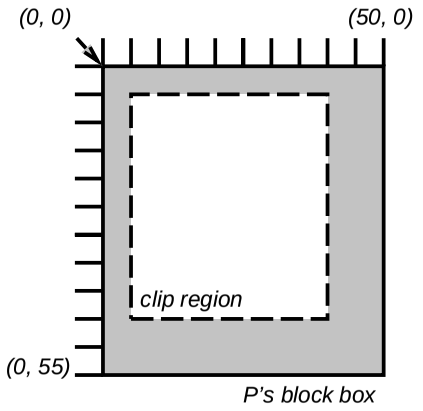
If the clipping region exceeds the bounds of the UA's document window, content may be clipped to that window by the native operating environment.

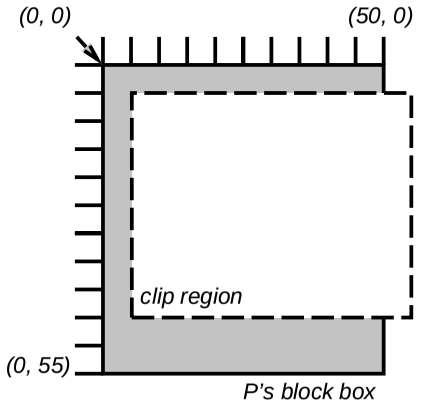
Example(s):

The following two rules:

```
p { clip: rect(5px, 40px, 45px, 5px); }
p { clip: rect(5px, 55px, 45px, 5px); }
```

will create the rectangular clipping regions delimited by the dashed lines in the following illustrations:





Note. *In CSS 2.1, all clipping regions are rectangular. We anticipate future extensions to permit non-rectangular clipping. Future versions may also reintroduce a syntax for offsetting shapes from each edge instead of offsetting from a point.*

11.2 Visibility: the 'visibility' property

'visibility'

Value:	visible	hidden	collapse	inherit
Initial:	visible
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

The 'visibility' property specifies whether the boxes generated by an element are rendered. Invisible boxes still affect layout (set the 'display' property to 'none' to suppress box generation altogether). Values have the following meanings:

visible
The generated box is visible.

hidden
The generated box is invisible (fully transparent, nothing is drawn), but still affects layout. Furthermore, descendents of the element will be visible if they have 'visibility: visible'.

collapse
Please consult the section on dynamic row and column effects [p. 251] in tables. If used on elements other than rows, row groups, columns, or column groups, 'collapse' has the same meaning as 'hidden'.

This property may be used in conjunction with scripts to create dynamic effects.

In the following example, pressing either form button invokes a user-defined script function that causes the corresponding box to become visible and the other to be hidden. Since these boxes have the same size and position, the effect is that one replaces the other. (The script code is in a hypothetical script language. It may or may not have any effect in a CSS-capable UA.)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<HTML>
<HEAD><TITLE>Dynamic visibility example</TITLE>
<META
 http-equiv="Content-Script-Type"
 content="application/x-hypothetical-scripting-language">
<STYLE type="text/css">
<!--
   #container1 { position: absolute;
                 top: 2in; left: 2in; width: 2in }
   #container2 { position: absolute;
                 top: 2in; left: 2in; width: 2in;
                 visibility: hidden; }
-->
</STYLE>
</HEAD>
<BODY>
<P>Choose a suspect:</P>
<DIV id="container1">
```



```
   <IMG alt="Al Capone"
        width="100" height="100"
        src="suspect1.png">
   <P>Name: Al Capone</P>
   <P>Residence: Chicago</P>
</DIV>

<DIV id="container2">
   <IMG alt="Lucky Luciano"
        width="100" height="100"
        src="suspect2.png">
   <P>Name: Lucky Luciano</P>
   <P>Residence: New York</P>
</DIV>

<FORM method="post"
      action="http://www.suspect.org/process-bums">
   <P>
   <INPUT name="Capone" type="button"
          value="Capone"
          onclick='show("container1");hide("container2")'>
   <INPUT name="Luciano" type="button"
          value="Luciano"
          onclick='show("container2");hide("container1")'>
</FORM>
</BODY>
</HTML>
```

12 Generated content, automatic numbering, and lists

Contents

In some cases, authors may want user agents to render content that does not come from the document tree [p. 33] . One familiar example of this is a numbered list; the author does not want to list the numbers explicitly, he or she wants the user agent to generate them automatically. Similarly, authors may want the user agent to insert the word "Figure" before the caption of a figure, or "Chapter 7" before the seventh chapter title. For audio or braille in particular, user agents should be able to insert these strings.

In CSS 2.1, content may be generated by two mechanisms:

- The 'content' property, in conjunction with the :before and :after pseudo-elements.
- Elements with a value of 'list-item' for the 'display' property.

12.1 The :before and :after pseudo-elements

Authors specify the style and location of generated content with the :before and :after pseudo-elements. As their names indicate, the :before and :after pseudo-elements specify the location of content before and after an element's document tree [p. 33] content. The 'content' property, in conjunction with these pseudo-elements, specifies what is inserted.

Example(s):

For example, the following rule inserts the string "Note: " before the content of every P element whose "class" attribute has the value "note":


```
p.note:before { content: "Note: " }
```

The formatting objects (e.g., boxes) generated by an element include generated content. So, for example, changing the above style sheet to:

```
p.note:before { content: "Note: " }
p.note        { border: solid green }
```

would cause a solid green border to be rendered around the entire paragraph, including the initial string.

The :before and :after pseudo-elements inherit [p. 80] any inheritable properties from the element in the document tree to which they are attached.

Example(s):

For example, the following rules insert an open quote mark before every Q element. The color of the quote mark will be red, but the font will be the same as the font of the rest of the Q element:

```
q:before {
  content: open-quote;
  color: red
}
```

In a :before or :after pseudo-element declaration, non-inherited properties take their initial values [p. 19] .

Example(s):

So, for example, because the initial value of the 'display' property is 'inline', the quote in the previous example is inserted as an inline box (i.e., on the same line as the element's initial text content). The next example explicitly sets the 'display' property to 'block', so that the inserted text becomes a block:

```
body:after {
    content: "The End";
    display: block;
    margin-top: 2em;
    text-align: center;
}
```

The :before and :after pseudo-elements elements interact with other boxes, such as run-in boxes, as if they were real elements inserted just inside their associated element.

Example(s):

For example, the following document fragment and stylesheet:

```
<h2> Header </h2>
<p> Text </p>
```

```
h2 { display: run-in; }
p:before { display: block; content: 'Some'; }
```

...would render in exactly the same way as the following document fragment and stylesheet:

```
<h2> Header </h2>            h2 { display: run-in; }
<p><span>Some</span> Text </p>  span { display: block }
```

Similarly, the following document fragment and stylesheet:

```
<h2> Header </h2>     h2 { display: run-in; }
                      h2:after { display: block; content: 'Thing'; }
<p> Text </p>
```

...would render in exactly the same way as the following document fragment and stylesheet:

```
<h2> Header <span>Thing</span></h2>   h2 { display: block; }
                                      span { display: block; }
<p> Text </p>
```

12.2 The 'content' property

'content'

Value:	normal I [<string> I <counter> I attr(<identifier>) I open-quote I close-quote I no-open-quote I no-close-quote]+ I inherit
Initial:	normal
Applies to:	:before and :after pseudo-elements
Inherited:	no
Percentages:	N/A
Media:	all
Computed value:	for URI values, the absolute URI; for attr() values, the resulting string; otherwise as specified

This property is used with the :before and :after pseudo-elements to generate content in a document. Values have the following meanings:

normal
The pseudo-element is not generated.

<string>
Text content (see the section on strings [p. 54]).

<uri>
The value is a URI that designates an external resource. If a user agent cannot support the resource because of the media types [p. 87] it supports, it must ignore the resource.

<counter>
Counters [p. 52] may be specified with two different functions: 'counter()' or 'counters()'. The former has two forms: 'counter(*name*)' or 'counter(*name*, *style*)'. The generated text is the value of the named counter at this point in the formatting structure; it is formatted in the indicated style [p. 187] ('decimal' by default). The latter function also has two forms: 'counters(*name*, *string*)' or 'counters(*name*, *string*, *style*)'. The generated text is the value of all counters with the given name at this point in the formatting structure, separated by the specified string. The counters are rendered in the indicated style [p. 187] ('decimal' by default). See the section on automatic counters and numbering [p. 184] for more information.

open-quote and **close-quote**
These values are replaced by the appropriate string from the **'quotes'** property.

no-open-quote and **no-close-quote**
Same as 'none', but increments (decrements) the level of nesting for quotes.

attr(X)
This function returns as a string the value of attribute X for the subject of the selector. The string is not parsed by the CSS processor. If the subject of the selector doesn't have an attribute X, an empty string is returned. The case-sensitivity of attribute names depends on the document language. **Note.** In CSS 2.1, it is not possible to refer to attribute values for other elements than the subject of the selector.

The 'display' property controls whether the content is placed in a block, inline, or marker box.

Example(s):

The following rule causes the string "Chapter: " to be generated before each H1 element:

```
H1:before {
  content: "Chapter: ";
  display: inline;
}
```

Authors may include newlines in the generated content by writing the "\A" escape sequence in one of the strings after the 'content' property. This inserted line break is still subject to the 'white-space' property. See "Strings" [p. 54] and "Characters and case" [p. 42] for more information on the "\A" escape sequence.

Example(s):

```
h1:before {
    display: block;
    text-align: center;
    content: "chapter\A hoofdstuk\A chapitre"
}
```

Generated content does not alter the document tree. In particular, it is not fed back to the document language processor (e.g., for reparsing).

12.3 Quotation marks

In CSS 2.1, authors may specify, in a style-sensitive and context-dependent manner, how user agents should render quotation marks. The 'quotes' property specifies pairs of quotation marks for each level of embedded quotation. The 'content' property gives access to those quotation marks and causes them to be inserted before and after a quotation.

12.3.1 Specifying quotes with the 'quotes' property

'quotes'

Value:	[<string> <string>]+ I none I inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies quotation marks for any number of embedded quotations. Values have the following meanings:

none
The 'open-quote' and 'close-quote' values of the 'content' property produce no quotation marks.

[**<string> <string>**]+
Values for the 'open-quote' and 'close-quote' values of the 'content' property are taken from this list of pairs of quotation marks (opening and closing). The first (leftmost) pair represents the outermost level of quotation, the second pair the first level of embedding, etc. The user agent must apply the appropriate pair of quotation marks according to the level of embedding.

Example(s):

For example, applying the following style sheet:

```
/* Specify pairs of quotes for two levels in two languages */
q:lang(en) { quotes: '"' '"' "'" "'" }
q:lang(no) { quotes: "«" "»" '"' '"' }

/* Insert quotes before and after Q element content */
q:before { content: open-quote }
q:after  { content: close-quote }
```

to the following HTML fragment:

```
<HTML lang="en">
  <HEAD>
  <TITLE>Quotes</TITLE>
  </HEAD>
  <BODY>
    <P><Q>Quote me!</Q>
  </BODY>
</HTML>
```

would allow a user agent to produce:


```
"Quote me!"
```

while this HTML fragment:

```
<HTML lang="no">
  <HEAD>
  <TITLE>Quotes</TITLE>
  </HEAD>
  <BODY>
    <P><Q>Trøndere gråter når <Q>Vinsjan på kaia</Q> blir deklamert.</Q>
  </BODY>
</HTML>
```

would produce:

```
«Trøndere gråter når "Vinsjan på kaia" blir deklamert.»
```

Note. *While the quotation marks specified by 'quotes' in the previous examples are conveniently located on computer keyboards, high quality typesetting would require different ISO 10646 characters. The following informative table lists some of the ISO 10646 quotation mark characters:*

Character	Approximate rendering	ISO 10646 code (hex)	Description
"	"	0022	QUOTATION MARK [the ASCII double quotation mark]
’	’	0027	APOSTROPHE [the ASCII single quotation mark]
‹	<	2039	SINGLE LEFT-POINTING ANGLE QUOTATION MARK
&#8250;	>	203A	SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
«	«	00AB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
»	»	00BB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
&#8216;	‘	2018	LEFT SINGLE QUOTATION MARK [single high-6]
&#8217;	’	2019	RIGHT SINGLE QUOTATION MARK [single high-9]
&#8220;	“	201C	LEFT DOUBLE QUOTATION MARK [double high-6]
&#8221;	”	201D	RIGHT DOUBLE QUOTATION MARK [double high-9]
&#8222;	,,	201E	DOUBLE LOW-9 QUOTATION MARK [double low-9]

12.3.2 Inserting quotes with the ’content’ property

Quotation marks are inserted in appropriate places in a document with the ’open-quote’ and ’close-quote’ values of the ’content’ property. Each occurrence of ’open-quote’ or ’close-quote’ is replaced by one of the strings from the value of ’quotes’, based on the depth of nesting.

’Open-quote’ refers to the first of a pair of quotes, ’close-quote’ refers to the second. Which pair of quotes is used depends on the nesting level of quotes: the number of occurrences of ’open-quote’ in all generated text before the current occurrence, minus the number of occurrences of ’close-quote’. If the depth is 0, the first pair is used, if the depth is 1, the second pair is used, etc. If the depth is greater than the number of pairs, the last pair is repeated. A ’close-quote’ that would make the depth negative is in error and is ignored (at rendering time): the depth stays at 0 and

no quote mark is rendered (although the rest of the ’content’ property’s value is still inserted).

Note. *The quoting depth is independent of the nesting of the source document or the formatting structure.*

Some typographic styles require open quotation marks to be repeated before every paragraph of a quote spanning several paragraphs, but only the last paragraph ends with a closing quotation mark. In CSS, this can be achieved by inserting "phantom" closing quotes. The keyword ’no-close-quote’ decrements the quoting level, but does not insert a quotation mark.

Example(s):

The following style sheet puts opening quotation marks on every paragraph in a BLOCKQUOTE, and inserts a single closing quote at the end:

```
blockquote p:before     { content: open-quote }
blockquote p:after      { content: no-close-quote }
blockquote p.last:after { content: close-quote }
```

This relies on the last paragraph being marked with a class "last".

For symmetry, there is also a ’no-open-quote’ keyword, which inserts nothing, but increments the quotation depth by one.

12.4 Automatic counters and numbering

Automatic numbering in CSS2 is controlled with two properties, ’counter-increment’ and ’counter-reset’. The counters defined by these properties are used with the counter() and counters() functions of the the ’content’ property.

’counter-reset’

Value:	[<identifier> <integer>?]+ I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all
Computed value:	as specified

’counter-increment’

Value:	[<identifier> <integer>?]+ I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	all
Computed value:	as specified

The 'counter-increment' property accepts one or more names of counters (identifiers), each one optionally followed by an integer. The integer indicates by how much the counter is incremented for every occurrence of the element. The default increment is 1. Zero and negative integers are allowed.

The 'counter-reset' property also contains a list of one or more names of counters, each one optionally followed by an integer. The integer gives the value that the counter is set to on each occurrence of the element. The default is 0.

If 'counter-increment' refers to a counter that is not in the scope (see below [p. 186]) of any 'counter-reset', the counter is assumed to have been reset to 0 by the root element.

Example(s):

This example shows a way to number chapters and sections with "Chapter 1", "1.1", "1.2", etc.

```
H1:before {
    content: "Chapter " counter(chapter) ". ";
    counter-increment: chapter;  /* Add 1 to chapter */
    counter-reset: section;      /* Set section to 0 */
}
H2:before {
    content: counter(chapter) "." counter(section) " ";
    counter-increment: section;
}
```

If an element increments/resets a counter and also uses it (in the 'content' property of its :before or :after pseudo-element), the counter is used *after* being incremented/reset.

If an element both resets and increments a counter, the counter is reset first and then incremented.

The 'counter-reset' property follows the cascading rules. Thus, due to cascading, the following style sheet:

```
H1 { counter-reset: section -1 }
H1 { counter-reset: imagenum 99 }
```

will only reset 'imagenum'. To reset both counters, they have to be specified together:


```
H1 { counter-reset: section -1 imagenum 99 }
```

12.4.1 Nested counters and scope

Counters are "self-nesting", in the sense that re-using a counter in a child element automatically creates a new instance of the counter. This is important for situations like lists in HTML, where elements can be nested inside themselves to arbitrary depth. It would be impossible to define uniquely named counters for each level.

Example(s):

Thus, the following suffices to number nested list items. The result is very similar to that of setting 'display:list-item' and 'list-style: inside' on the LI element:

```
OL { counter-reset: item }
LI { display: block }
LI:before { content: counter(item) ". "; counter-increment: item }
```

The self-nesting is based on the principle that every element that has a 'counter-reset' for a counter X, creates a fresh counter X, the *scope* of which is the element, its following siblings, and all the descendants of the element and its following siblings.

In the example above, an OL will create a counter, and all children of the OL will refer to that counter.

If we denote by item[n] the n^{th} instance of the "item" counter, and by "(" and ")" the beginning and end of a scope, then the following HTML fragment will use the indicated counters. (We assume the style sheet as given in the example above).

```
<OL>               <!-- (set item[0] to 0          -->
  <LI>item         <!--  increment item[0] (= 1)   -->
  <LI>item         <!--  increment item[0] (= 2)   -->
    <OL>           <!--  (set item[1] to 0         -->
      <LI>item     <!--   increment item[1] (= 1)  -->
      <LI>item     <!--   increment item[1] (= 2)  -->
      <LI>item     <!--   increment item[1] (= 3)  -->
        <OL>       <!--   (set item[2] to 0        -->
          <LI>item <!--    increment item[2] (= 1) -->
        </OL>      <!--   )                       -->
        <OL>       <!--   (set item[3] to 0        -->
          <LI>     <!--    increment item[3] (= 1) -->
        </OL>      <!--   )                       -->
      <LI>item     <!--   increment item[1] (= 4)  -->
    </OL>          <!--  )                        -->
  <LI>item         <!--  increment item[0] (= 3)   -->
  <LI>item         <!--  increment item[0] (= 4)   -->
</OL>              <!-- )                         -->
<OL>               <!-- (reset item[4] to 0       -->
  <LI>item         <!--  increment item[4] (= 1)   -->
  <LI>item         <!--  increment item[4] (= 2)   -->
</OL>              <!-- )                         -->
```

The ’counters()’ function generates a string composed of the values of all counters with the same name, separated by a given string.

Example(s):

The following style sheet numbers nested list items as "1", "1.1", "1.1.1", etc.

```
OL { counter-reset: item }
LI { display: block }
LI:before { content: counters(item, "."); counter-increment: item }
```

12.4.2 Counter styles

By default, counters are formatted with decimal numbers, but all the styles available for the ’list-style-type’ property are also available for counters. The notation is:

```
counter(name)
```

for the default style, or:

```
counter(name, 'list-style-type')
```

All the styles are allowed, including ’disc’, ’circle’, ’square’, and ’none’.

Example(s):

```
H1:before        { content: counter(chno, upper-latin) ". " }
H2:before        { content: counter(section, upper-roman) " - " }
BLOCKQUOTE:after { content: " [" counter(bq, hebrew) "]" }
DIV.note:before  { content: counter(notecntr, disc) " " }
P:before         { content: counter(p, none) }
```

12.4.3 Counters in elements with ’display: none’

An element that is not displayed (’display’ set to ’none’) cannot increment or reset a counter.

Example(s):

For example, with the following style sheet, H2s with class "secret" do not increment ’count2’.

```
H2.secret {counter-increment: count2; display: none}
```

Elements with ’visibility’ set to ’hidden’, on the other hand, *do* increment counters.

12.5 Lists

CSS 2.1 offers basic visual formatting of lists. An element with ’display: list-item’ generates a principal box [p. 109] for the element’s content and an optional marker box as a visual indication that the element is a list item.

The *list properties* describe basic visual formatting of lists: they allow style sheets to specify the marker type (image, glyph, or number), and the marker position with respect to the principal box (outside it or within it before content). They do not allow authors to specify distinct style (colors, fonts, alignment, etc.) for the list marker or adjust its position with respect to the principal box, these may be derived from the principal box.

The background properties [p. 206] apply to the principal box only; an ’outside’ marker box is transparent.

12.5.1 Lists: the ’list-style-type’, ’list-style-image’, ’list-style-position’, and ’list-style’ properties

’list-style-type’

Value:	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-latin | upper-latin | none | inherit
Initial:	disc
Applies to:	elements with ’display: list-item’
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies appearance of the list item marker if ’list-style-image’ has the value ’none’ or if the image pointed to by the URI cannot be displayed. The value ’none’ specifies no marker, otherwise there are three types of marker: glyphs, numbering systems, and alphabetic systems.

Glyphs are specified with **disc**, **circle**, and **square**. Their exact rendering depends on the user agent.

Numbering systems are specified with:

decimal
Decimal numbers, beginning with 1.
decimal-leading-zero
Decimal numbers padded by initial zeros (e.g., 01, 02, 03, ..., 98, 99).
lower-roman
Lowercase roman numerals (i, ii, iii, iv, v, etc.).
upper-roman
Uppercase roman numerals (I, II, III, IV, V, etc.).

A user agent that does not recognize a numbering system should use ’decimal’.

Alphabetic systems are specified with:

lower-latin or **lower-alpha**
: Lowercase ascii letters (a, b, c, ... z).

upper-latin or **upper-alpha**
: Uppercase ascii letters (A, B, C, ... Z).

This specification does not define how alphabetic systems wrap at the end of the alphabet. For instance, after 26 list items, 'lower-latin' rendering is undefined. Therefore, for long lists, we recommend that authors specify true numbers.

For example, the following HTML document:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
   <HEAD>
     <TITLE>Lowercase latin numbering</TITLE>
     <STYLE type="text/css">
          ol { list-style-type: lower-roman }
     </STYLE>
  </HEAD>
  <BODY>
    <OL>
      <LI> This is the first item.
      <LI> This is the second item.
      <LI> This is the third item.
    </OL>
  </BODY>
</HTML>
```

might produce something like this:

```
  i This is the first item.
 ii This is the second item.
iii This is the third item.
```

The list marker alignment (here, right justified) depends on the user agent.

'list-style-image'

Value:	<uri> I none I inherit
Initial:	none
Applies to:	elements with 'display: list-item'
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	absolute URI or 'none'

This property sets the image that will be used as the list item marker. When the image is available, it will replace the marker set with the 'list-style-type' marker.

Example(s):

The following example sets the marker at the beginning of each list item to be the image "ellipse.png".

```
ul { list-style-image: url("http://png.com/ellipse.png") }
```

'list-style-position'

Value:	inside I outside I inherit
Initial:	outside
Applies to:	elements with 'display: list-item'
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies the position of the marker box in the principal block box. Values have the following meanings:

outside
: The marker box is outside the principal block box. CSS 2.1 does not specify the precise location of the marker box.

inside
: The marker box is the first inline box in the principal block box, after which the element's content flows.

For example:

```
<HTML>
  <HEAD>
    <TITLE>Comparison of inside/outside position</TITLE>
    <STYLE type="text/css">
      ul         { list-style: outside }
      ul.compact { list-style: inside }
    </STYLE>
  </HEAD>
  <BODY>
    <UL>
      <LI>first list item comes first
      <LI>second list item comes second
    </UL>

    <UL class="compact">
      <LI>first list item comes first
      <LI>second list item comes second
    </UL>
  </BODY>
</HTML>
```

The above example may be formatted as:

- first list item comes first
- second list item comes second

- first list item comes first
- second list item comes second

The left sides of the list item boxes are not affected by marker placement

In right-to-left text, the markers would have been on the right side of the box.

'list-style'

Value:	[<'list-style-type'> || <'list-style-position'> || <'list-style-image'>] | inherit
Initial:	see individual properties
Applies to:	elements with 'display: list-item'
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	see individual properties

The 'list-style' property is a shorthand notation for setting the three properties 'list-style-type', 'list-style-image', and 'list-style-position' at the same place in the style sheet.

Example(s):

```
ul { list-style: upper-roman inside }  /* Any "ul" element */
ul > li > ul { list-style: circle outside } /* Any "ul" child
                                              of an "li" child
                                              of a "ul" element */
```

Although authors may specify 'list-style' information directly on list item elements (e.g., "li" in HTML), they should do so with care. The following rules look similar, but the first declares a descendant selector [p. 62] and the second a (more specific) child selector. [p. 63]


```
ol.alpha li   { list-style: lower-alpha } /* Any "li" descendant of an "ol" */
ol.alpha > li { list-style: lower-alpha } /* Any "li" child of an "ol" */
```

Authors who use only the descendant selector [p. 62] may not achieve the results they expect. Consider the following rules:

```
<HTML>
  <HEAD>
    <TITLE>WARNING: Unexpected results due to cascade</TITLE>
    <STYLE type="text/css">
      ol.alpha li  { list-style: lower-alpha }
      ul li        { list-style: disc }
    </STYLE>
  </HEAD>
  <BODY>
    <OL class="alpha">
      <LI>level 1
      <UL>
         <LI>level 2
      </UL>
    </OL>
  </BODY>
</HTML>
```

The desired rendering would have level 1 list items with 'lower-alpha' labels and level 2 items with 'disc' labels. However, the cascading order [p. 83] will cause the first style rule (which includes specific class information) to mask the second. The following rules solve the problem by employing a child selector [p. 63] instead:

```
ol.alpha > li  { list-style: lower-alpha }
ul li   { list-style: disc }
```

Another solution would be to specify 'list-style' information only on the list type elements:

```
ol.alpha  { list-style: lower-alpha }
ul        { list-style: disc }
```

Inheritance will transfer the 'list-style' values from OL and UL elements to LI elements. This is the recommended way to specify list style information.

Example(s):

A URI value may be combined with any other value, as in:

```
ul { list-style: url("http://png.com/ellipse.png") disc }
```

In the example above, the 'disc' will be used when the image is unavailable.

A value of 'none' for the 'list-style' property sets both 'list-style-type' and 'list-style-image' to 'none':

```
ul { list-style: none }
```

The result is that no list-item marker is displayed.

13 Paged media

Contents

13.1 Introduction to paged media

Paged media (e.g., paper, transparencies, pages that are displayed on computer screens, etc.) differ from continuous media [p. 89] in that the content of the document is split into one or more discrete pages. To handle page breaks, CSS2 extends the visual formatting model [p. 107] as follows:

1. The page box [p. 196] extends the box model [p. 91] to allow authors to specify page margins.
2. The *page model* extends the visual formatting model [p. 107] to account for page breaks. [p. 199]

The CSS 2.1 page model specifies how a document is formatted within the page box [p. 196] . The page box does not necessarily correspond to the real *sheet* where the document will ultimately be rendered (paper, transparency, screen, etc.). The user agent is responsible for transferring the page box to the sheet. Transfer possibilities include:

- Transferring one page box to one sheet (e.g., single-sided printing).
- Transferring two page boxes to both sides of the same sheet (e.g., double-sided printing).
- Transferring N (small) page boxes to one sheet (called "n-up").
- Transferring one (large) page box to N x M sheets (called "tiling").
- Creating signatures. A signature is a group of pages printed on a sheet, which, when folded and trimmed like a book, appear in their proper sequence.
- Printing one document to several output trays.
- Outputting to a file.

13.2 Page boxes: the @page rule

The *page box* is a rectangular region that contains two areas:

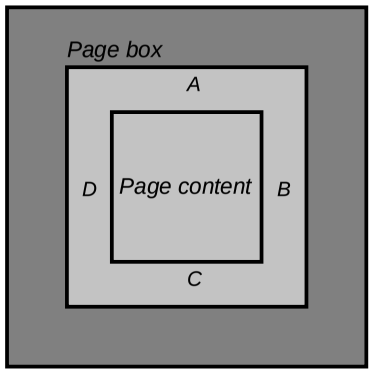
- The *page area*. The page area includes the boxes laid out on that page. The edges of the page area act as the initial containing block [p. 149] for layout that occurs between page breaks.
- The margin area, which surrounds the page area.

Authors can specify the margins of a page box inside an @page rule. An @page rule consists of the keyword "@page", followed by an optional page selector, followed by a block of declarations. The declarations are said to be in the *page context*.

The *page selector* specifies for which pages the declarations apply. In CSS 2.1, page selectors may designate the first page, all left pages, or all right pages

13.2.1 Page margins

The margin properties [p. 95] ('margin-top', 'margin-right', 'margin-bottom', 'margin-left', and 'margin') apply within the page context [p. 196] . The following diagram shows the relationships between the sheet, page box, and page margins:



A: margin−top
B: margin−right
C: margin−bottom
D: margin−left

Example(s):

Here is a simple example which sets all page margins on all pages:

```
@page {
  margin: 3cm;
}
```

The page context [p. 196] has no notion of fonts, so 'em' and 'ex' units are not allowed. Percentage values on the margin properties are relative to the dimensions of the page box [p. 196] ; for left and right margins, they refer to page box width while for top and bottom margins, they refer to page box height. All other units associated with the respective CSS 2.1 properties are allowed.

Due to negative margin values (either on the page box or on elements) or absolute positioning [p. 129] content may end up outside the page box, but this content may be "cut" — by the user agent, the printer, or ultimately, the paper cutter.

The computed value of box margins at the top or bottom of the page area is zero.

Rendering page boxes that do not fit a target sheet

If a page box does not fit the target sheet dimensions, the user agent may choose to:

- Rotate the page box 90° if this will make the page box fit.
- Scale the page to fit the target.

The user agent should consult the user before performing these operations.

Positioning the page box on the sheet

When the page box is smaller than the target size, the user agent is free to place the page box anywhere on the sheet. However, it is recommended that the page box be centered on the sheet since this will align double-sided pages and avoid accidental loss of information that is printed near the edge of the sheet.

13.2.2 Page selectors: selecting left, right, and first pages

When printing double-sided documents, the page boxes [p. 196] on left and right pages may be different. This can be expressed through two CSS pseudo-classes that may be used in page selectors.

All pages are automatically classified by user agents into either the :left or :right pseudo-class.

Example(s):


```
@page :left {
  margin-left: 4cm;
  margin-right: 3cm;
}

@page :right {
  margin-left: 3cm;
  margin-right: 4cm;
}
```

Authors may also specify style for the first page of a document with the :first pseudo-class:

Example(s):

```
@page { margin: 2cm } /* All margins set to 2cm */

@page :first {
  margin-top: 10cm    /* Top margin on first page 10cm */
}
```

Properties specified in a :left or :right @page rule override those specified in an @page rule that has no pseudo-class specified. Properties specified in a :first @page rule override those specified in :left or :right @page rules.

Margin declarations on left, right, and first pages may result in different page area [p. 196] widths. To simplify implementations, user agents may use a single page area width on left, right, and first pages. In this case, the page area width of the first page should be used.

13.2.3 Content outside the page box

When formatting content in the page model, some content may end up outside the page box. For example, an element whose 'white-space' property has the value 'pre' may generate a box that is wider than the page box. Also, when boxes are positioned absolutely [p. 129] , they may end up in "inconvenient" locations. For example, images may be placed on the edge of the page box or 100,000 meters below the page box.

The exact formatting of such elements lies outside the scope of this specification. However, we recommend that authors and user agents observe the following general principles concerning content outside the page box:

- Content should be allowed slightly beyond the page box to allow pages to "bleed".
- User agents should avoid generating a large number of empty page boxes to honor the positioning of elements (e.g., you don't want to print 100 blank pages).
- Authors should not position elements in inconvenient locations just to avoid rendering them.
- User agents may handle boxes positioned outside the page box in several ways, including discarding them or creating page boxes for them at the end of

the document.

13.3 Page breaks

This section describes page breaks in CSS 2.1. Five properties indicate where the user agent may or should break pages, and on what page (left or right) the subsequent content should resume. Each page break ends layout in the current page box [p. 196] and causes remaining pieces of the document tree [p. 33] to be laid out in a new page box.

13.3.1 Page break properties: 'page-break-before', 'page-break-after', 'page-break-inside'

'page-break-before'

Value:	auto I always I avoid I left I right I inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual, paged
Computed value:	as specified

'page-break-after'

Value:	auto I always I avoid I left I right I inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	no
Percentages:	N/A
Media:	visual, paged
Computed value:	as specified

'page-break-inside'

Value:	avoid I auto I inherit
Initial:	auto
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged
Computed value:	as specified

Values for these properties have the following meanings:

auto
Neither force nor forbid a page break before (after, inside) the generated box.

always
Always force a page break before (after) the generated box.

avoid
Avoid a page break before (after, inside) the generated box.

left
Force one or two page breaks before (after) the generated box so that the next page is formatted as a left page.

right
Force one or two page breaks before (after) the generated box so that the next page is formatted as a right page.

Whether the first page of a document is :left or :right depends on the major writing direction of the document. A conforming user agent may interpret the values 'left' and 'right' as 'always'.

A potential page break location is typically under the influence of the parent element's 'page-break-inside' property, the 'page-break-after' property of the preceding element, and the 'page-break-before' property of the following element. When these properties have values other than 'auto', the values 'always', 'left', and 'right' take precedence over 'avoid'.

These properties only apply to block level elements that are in the normal flow of the root element.

13.3.2 Breaks inside elements: 'orphans', 'widows'

'orphans'

Value:	<integer> I inherit
Initial:	2
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged
Computed value:	as specified

'widows'

Value:	<integer> I inherit
Initial:	2
Applies to:	block-level elements
Inherited:	yes
Percentages:	N/A
Media:	visual, paged
Computed value:	as specified

The 'orphans' property specifies the minimum number of lines of a paragraph that must be left at the bottom of a page. The 'widows' property specifies the minimum number of lines of a paragraph that must be left at the top of a page. Examples of how they are used to control page breaks are given below.

For information about paragraph formatting, please consult the section on line boxes [p. 118] .

13.3.3 Allowed page breaks

In the normal flow, page breaks can occur at the following places:

1. In the vertical margin between block boxes. When a page break occurs here, the computed values [p. 80] of the relevant 'margin-top' and 'margin-bottom' properties are set to '0'.
2. Between line boxes [p. 118] inside a block [p. 109] box.

These breaks are subject to the following rules:

- **Rule A:** Breaking at (1) is allowed only if the 'page-break-after' and 'page-break-before' properties of all the elements generating boxes that meet at this margin allow it, which is when at least one of them has the value 'always', 'left', or 'right', or when all of them are 'auto'.
- **Rule B:** However, if all of them are 'auto' and the nearest common ancestor of all the elements has a 'page-break-inside' value of 'avoid', then breaking here is not allowed.
- **Rule C:** Breaking at (2) is allowed only if the number of line boxes [p. 118] between the break and the start of the enclosing block box is the value of 'orphans' or more, and the number of line boxes between the break and the end of the box is the value of 'widows' or more.
- **Rule D:** In addition, breaking at (2) is allowed only if the 'page-break-inside' property is 'auto'.

If the above doesn't provide enough break points to keep content from overflowing the page boxes, then rules B and D are dropped in order to find additional break-points.

If that still does not lead to sufficient break points, rules A and C are dropped as well, to find still more break points.

13.3.4 Forced page breaks

A page break *must* occur at (1) if, among the 'page-break-after' and 'page-break-before' properties of all the elements generating boxes that meet at this margin, there is at least one with the value 'always', 'left', or 'right'.

13.3.5 "Best" page breaks

CSS2 does *not* define which of a set of allowed page breaks must be used; CSS2 does not forbid a user agent from breaking at every possible break point, or not to break at all. But CSS2 does recommend that user agents observe the following heuristics (while recognizing that they are sometimes contradictory):

- Break as few times as possible.
- Make all pages that don't end with a forced break appear to have about the same height.
- Avoid breaking inside a block that has a border.
- Avoid breaking inside a table.
- Avoid breaking inside a floated element

Example(s):

Suppose, for example, that the style sheet contains 'orphans: 4', 'widows: 2', and there are 20 lines (line boxes [p. 118]) available at the bottom of the current page:

- If a paragraph at the end of the current page contains 20 lines or fewer, it should be placed on the current page.
- If the paragraph contains 21 or 22 lines, the second part of the paragraph must not violate the 'widows' constraint, and so the second part must contain exactly two lines
- If the paragraph contains 23 lines or more, the first part should contain 20 lines and the second part the remaining lines.

Now suppose that 'orphans' is '10', 'widows' is '20', and there are 8 lines available at the bottom of the current page:

- If a paragraph at the end of the current page contains 8 lines or fewer, it should be placed on the current page.
- If the paragraph contains 9 lines or more, it cannot be split (that would violate the orphan constraint), so it should move as a block to the next page.

13.4 Cascading in the page context

Declarations in the page context [p. 196] obey the cascade [p. 79] just like normal CSS2 declarations.

Example(s):

Consider the following example:

```
@page {
  margin-left: 3cm;
}

@page :left {
  margin-left: 4cm;
}
```

Due to the higher specificity [p. 83] of the pseudo-class selector, the left margin on left pages will be '4cm' and all other pages (i.e., the right pages) will have a left margin of '3cm'.

---

14 Colors and Backgrounds

Contents

CSS properties allow authors to specify the foreground color and background of an element. Backgrounds may be colors or images. Background properties allow authors to position a background image, repeat it, and declare whether it should be fixed with respect to the viewport [p. 108] or scrolled along with the document.

See the section on color units [p. 53] for the syntax of valid color values.

14.1 Foreground color: the 'color' property

'color'

Value:	<color> I inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property describes the foreground color of an element's text content. There are different ways to specify red:

Example(s):

```
em { color: red }              /* predefined color name */
em { color: rgb(255,0,0) }     /* RGB range 0-255   */
```

14.2 The background

Authors may specify the background of an element (i.e., its rendering surface) as either a color or an image. In terms of the box model [p. 91] , "background" refers to the background of the content [p. 91] , padding [p. 91] and border [p. 91] areas. Border colors and styles are set with the border properties [p. 100] . Margins are always transparent.

Background properties are not inherited, but the parent box's background will shine through by default because of the initial 'transparent' value on 'background-color'.

The background of the root element becomes the background of the canvas and covers the entire canvas [p. 28] , anchored at the same point as it would be if it was painted only for the root element itself. The root element does not paint this background again.

For HTML documents, however, we recommend that authors specify the background for the BODY element rather than the HTML element. User agents should observe the following precedence rules to fill in the background of the canvas of HTML documents: if the value of the 'background' property for the HTML element is different from 'transparent' then use it, else use the value of the 'background' property for the BODY element. If the resulting value is 'transparent', the rendering is undefined. This does not apply to XHTML documents.

According to these rules, the canvas underlying the following HTML document will have a "marble" background:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
    <TITLE>Setting the canvas background</TITLE>
    <STYLE type="text/css">
       BODY { background: url("http://example.com/marble.png") }
    </STYLE>
    <P>My background is marble.
```

Note that the rule for the BODY element will work even though the BODY tag has been omitted in the HTML source since the HTML parser will infer the missing tag.

Backgrounds of elements that form a stacking context (see the 'z-index' property) are painted at the bottom of the element's stacking context, below anything in that stacking context.

14.2.1 Background properties: 'background-color', 'background-image', 'background-repeat', 'background-attachment', 'background-position', and 'background'

'background-color'

Value:	<color> I transparent I inherit
Initial:	transparent
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property sets the background color of an element, either a <color> value or the keyword 'transparent', to make the underlying colors shine through.

Example(s):

```
h1 { background-color: #F00 }
```

'background-image'

Value:	<uri> I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	absolute URI

This property sets the background image of an element. When setting a background image, authors should also specify a background color that will be used when the image is unavailable. When the image is available, it is rendered on top of the background color. (Thus, the color is visible in the transparent parts of the image).

Values for this property are either <uri>, to specify the image, or 'none', when no image is used.

Example(s):

```
body { background-image: url("marble.png") }
p { background-image: none }
```

'background-repeat'

Value:	repeat I repeat-x I repeat-y I no-repeat I inherit
Initial:	repeat
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

If a background image is specified, this property specifies whether the image is repeated (tiled), and how. All tiling covers the content [p. 91] , padding [p. 91] and border [p. 91] areas of a box.

The tiling and positioning of the background-image on inline elements is undefined in this specification. A future level of CSS may define the tiling and positioning of the background-image on inline elements.

Values have the following meanings:

repeat
The image is repeated both horizontally and vertically.

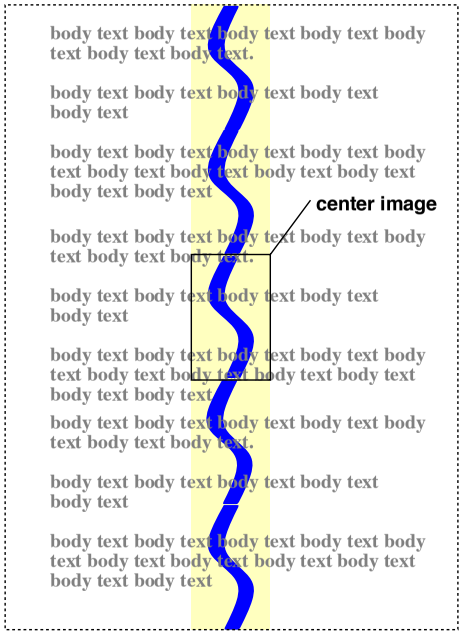
repeat-x
The image is repeated horizontally only.

repeat-y
The image is repeated vertically only.

no-repeat
The image is not repeated: only one copy of the image is drawn.

Example(s):

```
body {
  background: white url("pendant.png");
  background-repeat: repeat-y;
  background-position: center;
}
```



One copy of the background image is centered, and other copies are put above and below it to make a vertical band behind the element.

'background-attachment'

Value:	scroll I fixed I inherit
Initial:	scroll
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

If a background image is specified, this property specifies whether it is fixed with regard to the viewport [p. 108] ('fixed') or scrolls along with the containing block ('scroll').

Note that there is only one viewport per view. If an element has a scrolling mechanism (see 'overflow'), a 'fixed' background doesn't move with the element, and a 'scroll' background doesn't move with the scrolling mechanism.

Even if the image is fixed, it is still only visible when it is in the background, padding or border area of the element. Thus, unless the image is tiled ('background-repeat: repeat'), it may be invisible.

Example(s):

This example creates an infinite vertical band that remains "glued" to the viewport when the element is scrolled.

```
body {
  background: red url("pendant.png");
  background-repeat: repeat-y;
  background-attachment: fixed;
}
```

User agents that do not support 'fixed' backgrounds (for example due to limitations of the hardware platform) should ignore declarations with the keyword 'fixed'. For example:

```
body {
  background: white url(paper.png) scroll; /* for all UAs */
  background: white url(ledger.png) fixed; /* for UAs that do fixed backgrounds */
}
```

See the section on conformance [p. 34] for details.

'background-position'

Value:	[[<percentage> I <length> I top I center I bottom] II [<percentage> I <length> I left I center I right]] I inherit
Initial:	0% 0%
Applies to:	block-level and replaced elements
Inherited:	no
Percentages:	refer to the size of the box itself
Media:	visual
Computed value:	for <length> the absolute value, otherwise a percentage

If a background image has been specified, this property specifies its initial position. Values have the following meanings:

<percentage> <percentage>
With a value pair of '0% 0%', the upper left corner of the image is aligned with the upper left corner of the box's padding edge [p. 92] . A value pair of '100% 100%' places the lower right corner of the image in the lower right corner of padding area. With a value pair of '14% 84%', the point 14% across and 84% down the image is to be placed at the point 14% across and 84% down the padding area.

<length> <length>
With a value pair of '2cm 1cm', the upper left corner of the image is placed 2cm to the right and 1cm below the upper left corner of the padding area.

top left and **left top**
Same as '0% 0%'.

top, **top center**, and **center top**
Same as '50% 0%'.

right top and **top right**
Same as '100% 0%'.

left, **left center**, and **center left**
Same as '0% 50%'.

center and **center center**
Same as '50% 50%'.

right, **right center**, and **center right**
Same as '100% 50%'.

bottom left and **left bottom**
Same as '0% 100%'.

bottom, **bottom center**, and **center bottom**
Same as '50% 100%'.

bottom right and **right bottom**
Same as '100% 100%'.

If only one percentage or length value is given, it sets the horizontal position only, and the vertical position will be 50%. If two values are given, the horizontal position comes first. Combinations of keyword, length and percentage values are allowed, (e.g., '50% 2cm' or 'center 2cm' or 'center 10%'). For combinations of keyword and non-keyword values, 'left' and 'right' may only be used as the first value, and 'top' and 'bottom' may only be used as the second value. Negative positions are allowed.

The computed value of background-position for the purpose of inheritance is undefined, since the allowed values on this property may have different effects in a child element due to differences in size and position of their respective boxes.

Example(s):

```
body { background: url("banner.jpeg") right top }    /* 100%   0% */
body { background: url("banner.jpeg") top center }   /*  50%   0% */
body { background: url("banner.jpeg") center }       /*  50%  50% */
body { background: url("banner.jpeg") bottom }       /*  50% 100% */
```

If the background image is fixed within the viewport (see the 'background-attachment' property), the image is placed relative to the viewport instead of the element's padding area. For example,

Example(s):

```
body {
  background-image: url("logo.png");
  background-attachment: fixed;
  background-position: 100% 100%;
  background-repeat: no-repeat;
}
```

In the example above, the (single) image is placed in the lower-right corner of the viewport.

'background'

Value:	[<'background-color'> || <'background-image'> || <'background-repeat'> || <'background-attachment'> || <'background-position'>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	allowed on 'background-position'
Media:	visual
Computed value:	see individual properties

The 'background' property is a shorthand property for setting the individual background properties (i.e., 'background-color', 'background-image', 'background-repeat', 'background-attachment' and 'background-position') at the same place in the style sheet.

Given a valid declaration, the 'background' property first sets all the individual background properties to their initial values, then assigns explicit values given in the declaration.

Example(s):

In the first rule of the following example, only a value for 'background-color' has been given and the other individual properties are set to their initial value. In the second rule, all individual properties have been specified.

```
BODY { background: red }
P { background: url("chess.png") gray 50% repeat fixed }
```

14.3 Gamma correction

For information about gamma issues, please consult the Gamma Tutorial in the PNG specification ([PNG10]).

In the computation of gamma correction, UAs displaying on a CRT may assume an ideal CRT and ignore any effects on apparent gamma caused by dithering. That means the minimal handling they need to do on current platforms is:

PC using MS-Windows
    none
Unix using X11
    none
Mac using QuickDraw
    apply gamma 1.45 [ICC32] (ColorSync-savvy applications may simply pass the sRGB ICC profile to ColorSync to perform correct color correction)
SGI using X
    apply the gamma value from `/etc/config/system.glGammaVal` (the default value being 1.70; applications running on Irix 6.2 or above may simply pass the sRGB ICC profile to the color management system)
NeXT using NeXTStep
    apply gamma 2.22

"Applying gamma" means that each of the three R, G and B must be converted to $R'=R^{gamma}$, $G'=G^{gamma}$, $B'=B^{gamma}$, before being handed to the OS.

This may be done rapidly by building a 256-element lookup table once per browser invocation thus:

```
for i := 0 to 255 do
  raw := i / 255.0;
  corr := pow (raw, gamma);
  table[i] := trunc (0.5 + corr * 255.0)
end
```

which then avoids any need to do transcendental math per color attribute, far less per pixel.

15 Fonts

Contents

15.1 Introduction

Setting font properties will be among the most common uses of style sheets. Unfortunately, there exists no well-defined and universally accepted taxonomy for classifying fonts, and terms that apply to one font family may not be appropriate for others. E.g. 'italic' is commonly used to label slanted text, but slanted text may also be labeled as being *Oblique, Slanted, Incline, Cursive* or *Kursiv*. Therefore it is not a simple problem to map typical font selection properties to a specific font.

15.2 Font matching algorithm

Because there is no accepted, universal taxonomy of font properties, matching of properties to font faces must be done carefully. The properties are matched in a well-defined order to insure that the results of this matching process are as consistent as possible across UAs (assuming that the same library of font faces is presented to each of them).

1. The User Agent makes (or accesses) a database of relevant CSS 2.1 properties of all the fonts of which the UA is aware. If there are two fonts with exactly the same properties, the user agent selects one of them.
2. At a given element and for each character in that element, the UA assembles the font properties applicable to that element. Using the complete set of properties, the UA uses the 'font-family' property to choose a tentative font family. The remaining properties are tested against the family according to the matching criteria described with each property. If there are matches for all the remaining properties, then that is the matching font face for the given element.
3. If there is no matching font face within the 'font-family' being processed by step 2, and if there is a next alternative 'font-family' in the font set, then repeat step 2 with the next alternative 'font-family'.
4. If there is a matching font face, but it doesn't contain a glyph for the current character, and if there is a next alternative 'font-family' in the font sets, then repeat step 2 with the next alternative 'font-family'.
5. If there is no font within the family selected in 2, then use a UA-dependent default 'font-family' and repeat step 2, using the best match that can be obtained within the default font. If a particular character cannot be displayed using this font, then the UA has no suitable font for that character. The UA should map each character for which it has no suitable font to a visible symbol chosen by the UA, preferably a "missing character" glyph from one of the font faces available to the UA.

(The above algorithm can be optimized to avoid having to revisit the CSS 2.1 properties for each character.)

The per-property matching rules from (2) above are as follows:

1. 'font-style' is tried first. 'italic' will be satisfied if there is either a face in the UA's font database labeled with the CSS keyword 'italic' (preferred) or 'oblique'. Otherwise the values must be matched exactly or font-style will fail.
2. 'font-variant' is tried next. 'small-caps' matches (1) a font labeled as 'small-caps', (2) a font in which the small caps are synthesized, or (3) a font where all lowercase letters are replaced by upper case letters. A small-caps font may be synthesized by electronically scaling uppercase letters from a normal font. 'normal' matches a font's normal (non-small-caps) variant. A font cannot fail to have a normal variant. A font that is only available as small-caps shall be selectable as either a 'normal' face or a 'small-caps' face.
3. 'font-weight' is matched next, it will never fail. (See 'font-weight' below.)
4. 'font-size' must be matched within a UA-dependent margin of tolerance. (Typically, sizes for scalable fonts are rounded to the nearest whole pixel, while the tolerance for bitmapped fonts could be as large as 20%.) Further computations, e.g. by 'em' values in other properties, are based on the computed value of 'font-size'.

15.3 Font family: the 'font-family' property

'font-family'

Value:	[[<family-name> I <generic-family>] [, <family-name>I <generic-family>]*] I inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

The value is a prioritized list of font family names and/or generic family names. Unlike most other CSS properties, values are separated by a comma to indicate that they are alternatives:

```
body { font-family: Gill, Helvetica, sans-serif }
```

Although many fonts provide the "missing character" glyph, typically an open box, as its name implies this should not be considered a match for characters that cannot be found in the font. (It should, however, be considered a match for U+FFFD, the "missing character" character's code point).

There are two types of font family names:

<family-name>
: The name of a font family of choice. In the last example, "Gill" and "Helvetica" are font families.

<generic-family>
: In the example above, the last value is a generic family name. The following generic families are defined:

- 'serif' (e.g. Times)
- 'sans-serif' (e.g. Helvetica)
- 'cursive' (e.g. Zapf-Chancery)
- 'fantasy' (e.g. Western)
- 'monospace' (e.g. Courier)

Style sheet designers are encouraged to offer a generic font family as a last alternative. Generic font family names are keywords and must NOT be quoted.

If an unquoted font family name contains parentheses, brackets, and/or braces, they must still be either balanced or escaped per CSS grammar rules. Similarly, quote marks, semicolons, exclamation marks and commas within unquoted font family names must be escaped. Font names containing any such characters or whitespace should be quoted:

```
body { font-family: "New Century Schoolbook", serif }

<BODY STYLE="font-family: 'My own font', fantasy">
```

If quoting is omitted, any whitespace characters before and after the font name are ignored and any sequence of whitespace characters inside the font name is converted to a single space. Font family *names* that happen to be the same as a keyword value (e.g. 'initial', 'inherit', 'default', 'serif', 'sans-serif', 'monospace', 'fantasy', and 'cursive') must be quoted to prevent confusion with the keywords with the same names. UAs must not consider these keywords as matching the '<family-name>' type.

15.4 Font styling: the 'font-style' property

'font-style'

Value:	normal I italic I oblique I inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

The 'font-style' property selects between normal (sometimes referred to as "roman" or "upright"), italic and oblique faces within a font family.

A value of 'normal' selects a font that is classified as 'normal' in the UA's font database, while 'oblique' selects a font that is labeled 'oblique'. A value of 'italic' selects a font that is labeled 'italic', or, if that is not available, one labeled 'oblique'.

The font that is labeled 'oblique' in the UA's font database may actually have been generated by electronically slanting a normal font.

Fonts with Oblique, Slanted or Incline in their names will typically be labeled 'oblique' in the UA's font database. Fonts with *Italic, Cursive* or *Kursiv* in their names will typically be labeled 'italic'.

```
h1, h2, h3 { font-style: italic }
h1 em { font-style: normal }
```

In the example above, emphasized text within 'H1' will appear in a normal face.

15.5 Small-caps: the 'font-variant' property

'font-variant'

Value:	normal I small-caps I inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

Another type of variation within a font family is the small-caps. In a small-caps font the lower case letters look similar to the uppercase ones, but in a smaller size and with slightly different proportions. The 'font-variant' property selects that font.

A value of 'normal' selects a font that is not a small-caps font, 'small-caps' selects a small-caps font. It is acceptable (but not required) in CSS 2.1 if the small-caps font is a created by taking a normal font and replacing the lower case letters by scaled uppercase characters. As a last resort, uppercase letters will be used as replacement for a small-caps font.

The following example results in an 'H3' element in small-caps, with any emphasized words in oblique, and any emphasized words within an 'H3' oblique small-caps:

```
h3 { font-variant: small-caps }
em { font-style: oblique }
```

There may be other variants in the font family as well, such as fonts with old-style numerals, small-caps numerals, condensed or expanded letters, etc. CSS 2.1 has no properties that select those.

Note: insofar as this property causes text to be transformed to uppercase, the same considerations as for 'text-transform' apply.

15.6 Font boldness: the 'font-weight' property

'font-weight'

Value:	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	see text

The 'font-weight' property selects the weight of the font. The values '100' to '900' form an ordered sequence, where each number indicates a weight that is at least as dark as its predecessor. The keyword 'normal' is synonymous with '400', and 'bold' is synonymous with '700'. Keywords other than 'normal' and 'bold' have been shown to be often confused with font names and a numerical scale was therefore chosen for the 9-value list.

```
p { font-weight: normal }   /* 400 */
h1 { font-weight: 700 }     /* bold */
```

The 'bolder' and 'lighter' values select font weights that are relative to the weight inherited from the parent:

```
strong { font-weight: bolder }
```


Child elements inherit the resultant weight, not the keyword value.

Fonts (the font data) typically have one or more properties whose values are names that are descriptive of the "weight" of a font. There is no accepted, universal meaning to these weight names. Their primary role is to distinguish faces of differing darkness within a single font family. Usage across font families is quite variant; for example, a font that one might think of as being bold might be described as being *Regular, Roman, Book, Medium, Semi-* or *DemiBold, Bold,* or *Black,* depending on how black the "normal" face of the font is within the design. Because there is no standard usage of names, the weight property values in CSS 2.1 are given on a numerical scale in which the value '400' (or 'normal') corresponds to the "normal" text face for that family. The weight name associated with that face will typically be *Book, Regular, Roman, Normal* or sometimes *Medium*.

The association of other weights within a family to the numerical weight values is intended only to preserve the ordering of darkness within that family. However, the following heuristics tell how the assignment is done in typical cases:

- If the font family already uses a numerical scale with nine values (like e.g. *OpenType* does), the font weights should be mapped directly.
- If there is both a face labeled *Medium* and one labeled *Book, Regular, Roman* or *Normal,* then the *Medium* is normally assigned to the '500'.
- The font labeled "Bold" will often correspond to the weight value '700'.
- If there are fewer then 9 weights in the family, the default algorithm for filling the "holes" is as follows. If '500' is unassigned, it will be assigned the same font as '400'. If any of the values '600', '700', '800' or '900' remains unassigned, they are assigned to the same face as the next darker assigned keyword, if any, or the next lighter one otherwise. If any of '300', '200' or '100' remains unassigned, it is assigned to the next lighter assigned keyword, if any, or the next darker otherwise.

The following two examples show typical mappings.

Assume four weights in the "Rattlesnake" family, from lightest to darkest: *Regular, Medium, Bold, Heavy.*

First example of font-weight mapping

Available faces	Assignments	Filling the holes
"Rattlesnake Regular"	400	100, 200, 300
"Rattlesnake Medium"	500	
"Rattlesnake Bold"	700	600
"Rattlesnake Heavy"	800	900

Assume six weights in the "Ice Prawn" family: *Book, Medium, Bold, Heavy, Black, ExtraBlack.* Note that in this instance the user agent has decided *not* to assign a numeric value to "Ice Prawn ExtraBlack".

Second example of font-weight mapping

Available faces	Assignments	Filling the holes
"Ice Prawn Book"	400	100, 200, 300
"Ice Prawn Medium"	500	
"Ice Prawn Bold"	700	600
"Ice Prawn Heavy"	800	
"Ice Prawn Black"	900	
"Ice Prawn ExtraBlack"	(none)	

Since the intent of the relative keywords ’bolder’ and ’lighter’ is to darken or lighten the face *within the family* and because a family may not have faces aligned with all the symbolic weight values, the matching of ’bolder’ is to the next darker face available on the client within the family and the matching of ’lighter’ is to the next lighter face within the family. To be precise, the meaning of the relative keywords ’bolder’ and ’lighter’ is as follows:

- ’bolder’ selects the next weight that is assigned to a font that is darker than the inherited one. If there is no such weight, it simply results in the next darker numerical value (and the font remains unchanged), unless the inherited value was ’900’ in which case the resulting weight is also ’900’.
- ’lighter’ is similar, but works in the opposite direction: it selects the next lighter keyword with a different font from the inherited one, unless there is no such font, in which case it selects the next lighter numerical value (and keeps the font unchanged).

There is no guarantee that there will be a darker face for each of the ’font-weight’ values; for example, some fonts may have only a normal and a bold face, while others may have eight face weights. There is no guarantee on how a UA will map font faces within a family to weight values. The only guarantee is that a face of a given value will be no less dark than the faces of lighter values.

The computed value of "font-weight" is either:

- one of the legal number values, or
- one of the legal number values combined with one or more of the relative values (bolder or lighter). This type of computed values is necessary to use when the font in question does not have all weight variations that are needed.

CSS 2.1 does not specify how the computed value of font-weight is represented internally or externally.

15.7 Font size: the ’font-size’ property

’font-size’

Value:	<absolute-size>	<relative-size>	<length>	<percentage>	inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	refer to parent element’s font size
Media:	visual
Computed value:	absolute length

The font size corresponds to the em square, a concept used in typography. Note that certain glyphs may bleed outside their em squares. Values have the following meanings:

<absolute-size>

An <absolute-size> keyword is an index to a table of font sizes computed and kept by the UA. Possible values are:

[xx-small | x-small | small | medium | large | x-large | xx-large]

The following table provides user agent guidelines for the absolute-size scaling factor and their mapping to HTML heading and absolute font-sizes. The ’medium’ value is used as the reference middle value. The user agent may fine tune these values for different fonts or different types of display devices.

CSS absolute-size values	xx-small	x-small	small	medium	large	x-large	xx-large	
scaling factor	3/5	3/4	8/9	1	6/5	3/2	2/1	3/1
HTML headings	h6		h5	h4	h3	h2	h1	
HTML font sizes	1		2	3	4	5	6	7

Different media may need different scaling factors. Also, the UA should take the quality and availability of fonts into account when computing the table. The table may be different from one font family to another.

Note 1. *To preserve readability, a UA applying these guidelines should nevertheless avoid creating font-size resulting in less than 9 pixels per EM unit on a computer display.*

Note 2. *In CSS1, the suggested scaling factor between adjacent indexes was 1.5 which user experience proved to be too large. In CSS2, the suggested scaling factor for computer screen between adjacent indexes was 1.2 which still created issues for the small sizes. The new scaling factor varies between each index to provide better readability.*

<relative-size>

A <relative-size> keyword is interpreted relative to the table of font sizes and the font size of the parent element. Possible values are: [larger | smaller]. For example, if the parent element has a font size of 'medium', a value of 'larger' will make the font size of the current element be 'large'. If the parent element's size is not close to a table entry, the UA is free to interpolate between table entries or round off to the closest one. The UA may have to extrapolate table values if the numerical value goes beyond the keywords.

Length and percentage values should not take the font size table into account when calculating the font size of the element.

Negative values are not allowed.

On all other properties, 'em' and 'ex' length values refer to the computed font size of the current element. On the 'font-size' property, these length units refer to the computed font size of the parent element.

Note that an application may reinterpret an explicit size, depending on the context. E.g., inside a VR scene a font may get a different size because of perspective distortion.

Examples:

```
p { font-size: 16px; }
@media print {
        p { font-size: 12pt; }
}
blockquote { font-size: larger }
em { font-size: 150% }
em { font-size: 1.5em }
```

15.8 Shorthand font property: the 'font' property

'font'

Value:	[[<'font-style'> || <'font-variant'> || <'font-weight'>]? <'font-size'> [/ <'line-height'>]? <'font-family'>] | caption | icon | menu | message-box | small-caption | status-bar | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	yes
Percentages:	see individual properties
Media:	visual
Computed value:	see individual properties

The 'font' property is, except as described below [p. 223] , a shorthand property for setting 'font-style', 'font-variant', 'font-weight', 'font-size', 'line-height' and 'font-family' at the same place in the style sheet. The syntax of this property is based on a traditional typographical shorthand notation to set multiple properties related to fonts.

All font-related properties are first reset to their initial values, including those listed in the preceding paragraph. Then, those properties that are given explicit values in the 'font' shorthand are set to those values. For a definition of allowed and initial values, see the previously defined properties.

```
p { font: 12px/14px sans-serif }
p { font: 80% sans-serif }
p { font: x-large/110% "New Century Schoolbook", serif }
p { font: bold italic large Palatino, serif }
p { font: normal small-caps 120%/120% fantasy }
```

In the second rule, the font size percentage value ('80%') refers to the font size of the parent element. In the third rule, the line height percentage refers to the font size of the element itself.

In the first three rules above, the 'font-style', 'font-variant' and 'font-weight' are not explicitly mentioned, which means they are all three set to their initial value ('normal'). The fourth rule sets the 'font-weight' to 'bold', the 'font-style' to 'italic' and implicitly sets 'font-variant' to 'normal'.

The fifth rule sets the 'font-variant' ('small-caps'), the 'font-size' (120% of the parent's font), the 'line-height' (120% times the font size) and the 'font-family' ('fantasy'). It follows that the keyword 'normal' applies to the two remaining properties: 'font-style' and 'font-weight'.

The following values refer to system fonts:

caption
: The font used for captioned controls (e.g., buttons, drop-downs, etc.).

icon
: The font used to label icons.

menu
: The font used in menus (e.g., dropdown menus and menu lists).

message-box
: The font used in dialog boxes.

small-caption
: The font used for labeling small controls.

status-bar
: The font used in window status bars.

System fonts may only be set as a whole; that is, the font family, size, weight, style, etc. are all set at the same time. These values may then be altered individually if desired. If no font with the indicated characteristics exists on a given platform, the user agent should either intelligently substitute (e.g., a smaller version of the 'caption' font might be used for the 'small-caption' font), or substitute a user agent default font. As for regular fonts, if, for a system font, any of the individual properties are not part of the operating system's available user preferences, those properties should be set to their initial values.

That is why this property is "almost" a shorthand property: system fonts can only be specified with this property, not with 'font-family' itself, so 'font' allows authors to do more than the sum of its subproperties. However, the individual properties such as 'font-weight' are still given values taken from the system font, which can be independently varied.

Example(s):

```
button { font: 300 italic 1.3em/1.7em "FB Armada", sans-serif }
button p { font: menu }
button p em { font-weight: bolder }
```

If the font used for dropdown menus on a particular system happened to be, for example, 9-point Charcoal, with a weight of 600, then P elements that were descendants of BUTTON would be displayed as if this rule were in effect:

```
button p { font: 600 9px Charcoal }
```

Because the 'font' shorthand property resets any property not explicitly given a value to its initial value, this has the same effect as this declaration:

```
button p {
  font-family: Charcoal;
  font-style: normal;
  font-variant: normal;
  font-weight: 600;
  font-size: 9px;
  line-height: normal;
}
```

16 Text

Contents

The properties defined in the following sections affect the visual presentation of characters, spaces, words, and paragraphs.

16.1 Indentation: the 'text-indent' property

'text-indent'

Value:	<length>	<percentage>	inherit
Initial:	0
Applies to:	block-level elements, table cells and inline blocks
Inherited:	yes
Percentages:	refer to width of containing block
Media:	visual
Computed value:	the percentage as specified or the absolute length

This property specifies the indentation of the first line of text in a block. More precisely, it specifies the indentation of the first box that flows into the block's first line box [p. 118] . The box is indented with respect to the left (or right, for right-to-left layout) edge of the line box. User agents should render this indentation as blank space.

Values have the following meanings:

<length>
The indentation is a fixed length.
<percentage>
The indentation is a percentage of the containing block width.

The value of 'text-indent' may be negative, but there may be implementation-specific limits. If the value of 'text-indent' is either negative or exceeds the width of the block, that *first box*, described above, may overflow the block. The value of 'overflow' will affect whether such text that overflows the block is visible.

Example(s):

The following example causes a '3em' text indent.

```
p { text-indent: 3em }
```

Note: Since the 'text-indent' property inherits, when specified on a block element, it will affect descendent inline-block elements. For this reason, it is often wise to specify '`text-indent: 0`' on elements that are specified '`display:inline-block`'.

16.2 Alignment: the 'text-align' property

'text-align'

Value:	left	right	center	justify	inherit
Initial:	'left' if 'direction' is 'ltr'; 'right' if 'direction' is 'rtl'
Applies to:	block-level elements and table cells
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property describes how inline content of a block is aligned. Values have the following meanings:

left, right, center, justify
Left, right, center, and justify text, respectively.

A block of text is a stack of line boxes [p. 118] . In the case of 'left', 'right' and 'center', this property specifies how the inline boxes within each line box align with respect to the line box's left and right sides; alignment is not with respect to the viewport [p. 108] . In the case of 'justify', the UA may stretch the inline boxes in addition to adjusting their positions. (See also 'letter-spacing' and 'word-spacing'.)

If the computed value of text-align is 'justify' while the computed value of white-space is 'pre' or 'pre-line', the actual value of text-align is set to the initial value.

Example(s):

In this example, note that since 'text-align' is inherited, all block-level elements inside the DIV element with 'class=important' will have their inline content centered.

```
div.important { text-align: center }
```

Note. *The actual justification algorithm used depends on the user-agent and the language/script of the text.*

Conforming user agents [p. 34] may interpret the value 'justify' as 'left' or 'right', depending on whether the element's default writing direction is left-to-right or right-to-left, respectively.

16.3 Decoration

16.3.1 Underlining, overlining, striking, and blinking: the 'text-decoration' property

'text-decoration'

Value:	none I [underline II overline II line-through II blink] I inherit
Initial:	none
Applies to:	all elements
Inherited:	no (see prose)
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property describes decorations that are added to the text of an element. When specified on an inline element, it affects all the boxes generated by that element, otherwise, the decorations are propagated to the anonymous inline box that wraps all the inline contents of the element, using the element's color. It is not, however, further propagated to floating and absolutely positioned descendants, nor to the contents of 'inline-table' and 'inline-block' descendants. Nor is it propagated to block-level [p. 109] descendants of inline elements.

If an element contains no text (ignoring white space in elements that have 'white-space' set to 'normal', 'pre-line', or 'no-wrap'), user agents must refrain from rendering text decorations on the element. For example, elements containing only images and collapsed white space will not be underlined.

Text decorations on inline boxes are drawn across the entire element, going across any descendant elements without paying any attention to their presence. The 'text-decoration' property on descendant elements cannot have any effect on the decoration of the element. In determining the position of and thickness of text decoration lines, user agents may consider the font sizes of and dominant baselines of descendants, but must use the same baseline and thickness on each line.

Values have the following meanings:

none
: Produces no text decoration.

underline
: Each line of text is underlined.

overline
: Each line of text has a line above it.

line-through
: Each line of text has a line through the middle.

blink
: Text blinks (alternates between visible and invisible). Conforming user agents [p. 34] may simply not blink the text. Note that not blinking the text is one technique to satisfy checkpoint 3.3 of WAI-UAAG [p. ??] .

The color(s) required for the text decoration must be derived from the 'color' property value of the element on which 'text-decoration' is set. The color of decorations should remain the same even if descendant elements have different 'color' values.

Some user agents have implemented text-decoration by propagating the decoration to the descendant elements as opposed to simply drawing the decoration through the elements as described above. This was arguably allowed by the looser wording in CSS2. SVG1, CSS1-only, and CSS2-only user agents may implement the older model and still claim conformance to this part of CSS2.1. (This does not apply to UAs developed after this specification was released.)

Example(s):

In the following example for HTML, the text content of all A elements acting as hyperlinks (whether visited or not) will be underlined:

```
a:visited,a:link { text-decoration: underline }
```

Example(s):

In the following stylesheet and document fragment:

```
blockquote { text-decoration: underline; color: blue; }
em { display: block; }
cite { color: fuchsia; }

<blockquote>
 <p>
  <span>
   Help, help!
   <em> I am under a hat! </em>
   <cite> —GwieF </cite>
  </span>
 </p>
</blockquote>
```

...the underlining for the blockquote element is propagated to an anonymous inline element that surrounds the span element, causing the text "Help, help!" to be blue, with the blue underlining from the anonymous inline underneath it, the color being taken from the blockquote element. The `<em>text</em>` in the em block is not

underlined at all, as it is not contained in the same anonymous inline element. The final line of text is fuchsia, but the underline underneath it is still the blue underline from the anonymous inline element.

This diagram shows the boxes involved in the example above. The rounded aqua line represents the anonymous inline element wrapping the inline contents of the paragraph element, the rounded blue line represents the span element, and the orange lines represent the blocks.

16.4 Letter and word spacing: the 'letter-spacing' and 'word-spacing' properties

'letter-spacing'

Value:	normal | <length> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	'normal' or absolute length

This property specifies spacing behavior between text characters. Values have the following meanings:

normal
: The spacing is the normal spacing for the current font. This value allows the user agent to alter the space between characters in order to justify text.

<length>
: This value indicates inter-character space *in addition to* the default space between characters. Values may be negative, but there may be implementation-specific limits. User agents may not further increase or decrease the inter-character space in order to justify text.

Character spacing algorithms are user agent-dependent.

Example(s):

In this example, the space between characters in BLOCKQUOTE elements is increased by '0.1em'.

```
blockquote { letter-spacing: 0.1em }
```

In the following example, the user agent is not permitted to alter inter-character space:

```
blockquote { letter-spacing: 0cm }   /* Same as '0' */
```

When the resultant space between two characters is not the same as the default space, user agents should not use ligatures.

'word-spacing'

Value:	normal | <length> | inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	for 'normal' the value '0'; otherwise the absolute length

This property specifies spacing behavior between words. Values have the following meanings:

normal
: The normal inter-word space, as defined by the current font and/or the UA.

<length>
: This value indicates inter-word space *in addition to* the default space between words. Values may be negative, but there may be implementation-specific limits.

Word spacing algorithms are user agent-dependent. Word spacing is also influenced by justification (see the 'text-align' property).

Example(s):

In this example, the word-spacing between each word in H1 elements is increased by '1em'.

```
h1 { word-spacing: 1em }
```

16.5 Capitalization: the 'text-transform' property

'text-transform'

Value:	capitalize I uppercase I lowercase I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property controls capitalization effects of an element's text. Values have the following meanings:

capitalize
: Puts the first character of each word in uppercase.

uppercase
: Puts all characters of each word in uppercase.

lowercase
: Puts all characters of each word in lowercase.

none
: No capitalization effects.

The actual transformation in each case is written language dependent. See RFC 2070 ([RFC2070]) for ways to find the language of an element.

Conforming user agents [p. 34] may consider the value of 'text-transform' to be 'none' for characters that are not from the Latin-1 repertoire and for elements in languages for which the transformation is different from that specified by the case-conversion tables of ISO 10646 ([ISO10646]).

Example(s):

In this example, all text in an H1 element is transformed to uppercase text.

```
h1 { text-transform: uppercase }
```

16.6 Whitespace: the 'white-space' property

'white-space'

Value:	normal I pre I nowrap I pre-wrap I pre-line I inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property declares how whitespace [p. 40] inside the element is handled. Values have the following meanings:

normal
: This value directs user agents to collapse sequences of whitespace, and break lines as necessary to fill line boxes.

pre
: This value prevents user agents from collapsing sequences of whitespace. Lines are only broken at newlines in the source, or at occurrences of "\A" in generated content.

nowrap
: This value collapses whitespace as for 'normal', but suppresses line breaks within text.

pre-wrap
: This value prevents user agents from collapsing sequences of whitespace. Lines are broken at newlines in the source, at occurrences of "\A" in generated content, and as necessary to fill line boxes.

pre-line
: This value directs user agents to collapse sequences of whitespace. Lines are broken at newlines in the source, at occurrences of "\A" in generated content, and as necessary to fill line boxes.

Example(s):

The following examples show what whitespace [p. 40] behavior is expected from the PRE and P elements, the "nowrap" attribute in HTML, and in generated content.

```
pre         { white-space: pre }
p           { white-space: normal }
td[nowrap]  { white-space: nowrap }
:before,:after { white-space: pre-line }
```

In addition, the effect of an HTML PRE element with the *non-standard* "wrap" attribute is demonstrated by the following example:

```
pre[wrap]   { white-space: pre-wrap }
```

16.6.1 The 'white-space' processing model

Any text that is directly contained inside a block (not inside an inline) should be treated as an anonymous inline element.

For each inline (including anonymous inlines), the following steps are performed, ignoring bidi formatting characters as if they were not there:

1. Each non-linefeed whitespace character surrounding a linefeed character is removed if 'white-space' is set to 'normal', 'no-wrap', or 'pre-line'.
2. If 'white-space' is set to 'pre' or 'pre-wrap', any sequence of spaces (U+0020) unbroken by an element boundary is treated as a sequence of non-breaking spaces. However, a line breaking opportunity exists at the end of the sequence.
3. If 'white-space' is set to 'normal' or 'nowrap', linefeed characters are transformed for rendering purpose into one of the following characters: a space character, a zero width space character (U+200B), or no character (i.e. not rendered), according to UA-specific algorithms based on the content script.
4. If 'white-space' is set to 'normal', 'nowrap', or 'pre-line',
    1. every tab (U+0009) is converted to a space (U+0020)
    2. any space (U+0020) following another space (U+0020) — even a space before the inline, if that space also has 'white-space' set to 'normal', 'nowrap' or 'pre-line' — is removed.

Then, the entire block is rendered. Inlines are laid out, taking bidi reordering into account, and wrapping as specified by the 'white-space' property.

As each line is laid out,

1. If a space (U+0020) at the beginning of a line has 'white-space' set to 'normal', 'nowrap', or 'pre-line', it is removed.
2. All tabs (U+0009) are rendered as a horizontal shift that lines up the start edge of the next glyph with the next tab stop. Tab stops occur at points that are mutiples of 8 times the width of a space (U+0020) rendered in the block's font from the block's starting content edge.
3. If a space (U+0020) at the end of a line has 'white-space' set to 'normal', 'nowrap', or 'pre-line', it is also removed.

16.6.2 Example of bidirectionality with white-space collapsing

Given the following markup fragment, taking special note of spaces (with varied backgrounds and borders for emphasis and identification):

```
<ltr>A <rtl> B </rtl> C</ltr>
```

...where the `<ltr>` element represents a left-to-right embedding and the `<rtl>` element represents a right-to-left embedding, and assuming that the 'white-space' property is set to 'normal', the above processing model would result in the following:

- The space before the B ( ) would collapse with the space after the A ( ).
- The space before the C ( ) would collapse with the space after the B ( ).

This would leave two spaces, one after the A in the left-to-right embedding level, and one after the B in the right-to-left embedding level. This is then rendered according to the Unicode bidirectional algorithm, with the end result being:

```
A  BC
```

Note that there are two spaces between A and B, and none between B and C. This is best avoided by using the natural bidirectionality of characters instead of explicit embedding levels.

---

17 Tables

Contents

17.1 Introduction to tables

Table layout can be used to represent tabular relationships between data. Authors specify these relationships in the document language [p. 32] and can specify their *presentation* using CSS 2.1.

In a visual medium, CSS tables can also be used to achieve specific layouts. In this case, authors should not use table-related elements in the document language, but should apply the CSS to the relevant structural elements to achieve the desired layout.

Authors may specify the visual formatting of a table as a rectangular grid of cells. Rows and columns of cells may be organized into row groups and column groups. Rows, columns, row groups, column groups, and cells may have borders drawn around them (there are two border models in CSS 2.1). Authors may align data vertically or horizontally within a cell and align data in all cells of a row or column.

Example(s):

Here is a simple three-row, three-column table described in HTML 4.0:

```
<TABLE>
<CAPTION>This is a simple 3x3 table</CAPTION>
<TR id="row1">
   <TH>Header 1      <TD>Cell 1        <TD>Cell 2
<TR id="row2">
   <TH>Header 2      <TD>Cell 3        <TD>Cell 4
<TR id="row3">
   <TH>Header 3      <TD>Cell 5        <TD>Cell 6
</TABLE>
```

This code creates one table (the TABLE element), three rows (the TR elements), three header cells (the TH elements), and six data cells (the TD elements). Note that the three columns of this example are specified implicitly: there are as many columns in the table as required by header and data cells.

The following CSS rule centers the text horizontally in the header cells and presents the text in the header cells with a bold font weight

```
th { text-align: center; font-weight: bold }
```

The next rules align the text of the header cells on their baseline and vertically center the text in each data cell:

```
th { vertical-align: baseline }
td { vertical-align: middle }
```

The next rules specify that the top row will be surrounded by a 3px solid blue border and each of the other rows will be surrounded by a 1px solid black border:

```
table   { border-collapse: collapse }
tr#row1 { border-top: 3px solid blue }
tr#row2 { border-top: 1px solid black }
tr#row3 { border-top: 1px solid black }
```

Note, however, that the borders around the rows overlap where the rows meet. What color (black or blue) and thickness (1px or 3px) will the border between row1 and row2 be? We discuss this in the section on border conflict resolution. [p. 255]

The following rule puts the table caption above the table:

```
caption { caption-side: top }
```

The preceding example shows how CSS works with HTML 4.0 elements; in HTML 4.0, the semantics of the various table elements (TABLE, CAPTION, THEAD, TBODY, TFOOT, COL, COLGROUP, TH, and TD) are well-defined. In other document languages (such as XML applications), there may not be pre-defined table elements. Therefore, CSS 2.1 allows authors to "map" document language elements to table elements via the 'display' property. For example, the following rule makes the FOO element act like an HTML TABLE element and the BAR element act like a CAPTION element:

```
FOO { display : table }
BAR { display : table-caption }
```

We discuss the various table elements in the following section. In this specification, the term *table element* refers to any element involved in the creation of a table. An "internal" table element is one that produces a row, row group, column, column group, or cell.

17.2 The CSS table model

The CSS table model is based on the HTML 4.0 table model, in which the structure of a table closely parallels the visual layout of the table. In this model, a table consists of an optional caption and any number of rows of cells. The table model is said to be "row primary" since authors specify rows, not columns, explicitly in the document language. Columns are derived once all the rows have been specified -- the first cell of each row belongs to the first column, the second to the second column, etc.). Rows and columns may be grouped structurally and this grouping reflected in presentation (e.g., a border may be drawn around a group of rows).

Thus, the table model consists of tables, captions, rows, row groups, columns, column groups, and cells.

The CSS model does not require that the document language [p. 32] include elements that correspond to each of these components. For document languages (such as XML applications) that do not have pre-defined table elements, authors must map document language elements to table elements; this is done with the 'display' property. The following 'display' values assign table formatting rules to an arbitrary element:

table (In HTML: TABLE)
: Specifies that an element defines a block-level [p. 109] table: it is a rectangular block that participates in a block formatting context [p. 117] .

inline-table (In HTML: TABLE)
: Specifies that an element defines an inline-level [p. 111] table: it is a rectangular block that participates in an inline formatting context [p. 118]).

table-row (In HTML: TR)
: Specifies that an element is a row of cells.

table-row-group (In HTML: TBODY)
: Specifies that an element groups one or more rows.

table-header-group (In HTML: THEAD)
: Like 'table-row-group', but for visual formatting, the row group is always displayed before all other rows and rowgroups and after any top captions. Print user agents may repeat header rows on each page spanned by a table. Use of multiple elements with 'display: table-header-group' is undefined.

table-footer-group (In HTML: TFOOT)
: Like 'table-row-group', but for visual formatting, the row group is always displayed after all other rows and rowgroups and before any bottom captions. Print user agents may repeat footer rows on each page spanned by a table. Use of multiple elements with 'display: table-footer-group' is undefined.

table-column (In HTML: COL)
: Specifies that an element describes a column of cells.

table-column-group (In HTML: COLGROUP)
: Specifies that an element groups one or more columns.

table-cell (In HTML: TD, TH)
: Specifies that an element represents a table cell.

table-caption (In HTML: CAPTION)
: Specifies a caption for the table. Use of multiple elements with 'display: caption' is undefined; authors should not put more than one element with 'display: caption' inside a table or inline-table element.

Elements with 'display' set to 'table-column' or 'table-column-group' are not rendered (exactly as if they had 'display: none'), but they are useful, because they may have attributes which induce a certain style for the columns they represent.

The default style sheet for HTML 4.0 [p. 293] in the appendix illustrates the use of these values for HTML 4.0:

```
table    { display: table }
tr       { display: table-row }
thead    { display: table-header-group }
tbody    { display: table-row-group }
tfoot    { display: table-footer-group }
col      { display: table-column }
colgroup { display: table-column-group }
td, th   { display: table-cell }
caption  { display: table-caption }
```

User agents may ignore [p. 46] these 'display' property values for HTML table elements, since HTML tables may be rendered using other algorithms intended for backwards compatible rendering. However, this is not meant to discourage the use of 'display: table' on other, non-table elements in HTML.

17.2.1 Anonymous table objects

Document languages other than HTML may not contain all the elements in the CSS 2.1 table model. In these cases, the "missing" elements must be assumed in order for the table model to work. Any table element will automatically generate necessary anonymous table objects around itself, consisting of at least three nested objects corresponding to a 'table'/'inline-table' element, a 'table-row' element, and a 'table-cell' element. Missing elements generate anonymous [p. 111] objects (e.g., anonymous boxes in visual table layout) according to the following rules:

1. If the parent P of a 'table-cell' element T is not a 'table-row', an object corresponding to a 'table-row' will be generated between P and T. This object will span all consecutive 'table-cell' siblings (in the document tree) of T.
2. If the parent P of a 'table-row' element T is not a 'table', 'inline-table', or 'table-row-group' element, an object corresponding to a 'table' element will be generated between P and T. This object will span all consecutive siblings (in the

document tree) of T that require a 'table' parent: 'table-row', 'table-row-group', 'table-header-group', 'table-footer-group', 'table-column', 'table-column-group', and 'table-caption'. T and T's siblings may also be anonymous 'table-row' objects generated by rule 1.

3. If the parent P of a 'table-column' element T is not a 'table', 'inline-table', or 'table-column-group' element, an object corresponding to a 'table' element will be generated between P and T. This object will span all consecutive siblings (in the document tree) of T that require a 'table' parent: 'table-row', 'table-row-group', 'table-header-group', 'table-footer-group', 'table-column', 'table-column-group', and 'table-caption', including any anonymous 'table-row' objects generated by rule 1.
4. If the parent P of a 'table-row-group' (or 'table-header-group', 'table-footer-group', or 'table-column-group' or 'table-caption') element T is not a 'table' or 'inline-table', an object corresponding to a 'table' element will be generated between P and T. This object will span all consecutive siblings (in the document tree) of T that require a 'table' parent: 'table-row', 'table-row-group', 'table-header-group', 'table-footer-group', 'table-column', 'table-column-group', and 'table-caption', including any anonymous 'table-row' objects generated by rule 1.
5. If a child T of a 'table' element (or 'inline-table') P is not a 'table-row-group', 'table-header-group', 'table-footer-group', or 'table-row' element, an object corresponding to a 'table-row' element will be generated between P and T. This object spans all consecutive siblings of T that are not 'table-row-group', 'table-header-group', 'table-footer-group', or 'table-row' elements.
6. If a child T of a 'table-row-group' element (or 'table-header-group' or 'table-footer-group') P is not a 'table-row' element, an object corresponding to a 'table-row' element will be generated between P and T. This object spans all consecutive siblings of T that are not 'table-row' elements.
7. If a child T of a 'table-row' element P is not a 'table-cell' element, an object corresponding to a 'table-cell' element will be generated between P and T. This object spans all consecutive siblings of T that are not 'table-cell' elements.

Example(s):

In this XML example, a 'table' element is assumed to contain the HBOX element:

```
<HBOX>
  <VBOX>George</VBOX>
  <VBOX>4287</VBOX>
  <VBOX>1998</VBOX>
</HBOX>
```

because the associated style sheet is:

```
HBOX { display: table-row }
VBOX { display: table-cell }
```


Example(s):

In this example, three 'table-cell' elements are assumed to contain the text in the ROWs. Note that the text is further encapsulated in anonymous inline boxes, as explained in visual formatting model [p. 111] :

```
<STACK>
  <ROW>This is the <D>top</D> row.</ROW>
  <ROW>This is the <D>middle</D> row.</ROW>
  <ROW>This is the <D>bottom</D> row.</ROW>
</STACK>
```

The style sheet is:

```
STACK { display: inline-table }
ROW   { display: table-row }
D     { display: inline; font-weight: bolder }
```

17.3 Columns

Table cells may belong to two contexts: rows and columns. However, in the source document cells are descendants of rows, never of columns. Nevertheless, some aspects of cells can be influenced by setting properties on columns.

The following properties apply to column and column-group elements:

'border'
: The various border properties apply to columns only if 'border-collapse' is set to 'collapse' on the table element. In that case, borders set on columns and column groups are input to the conflict resolution algorithm [p. 255] that selects the border styles at every cell edge.

'background'
: The background properties set the background for cells in the column, but only if both the cell and row have transparent backgrounds. See "Table layers and transparency." [p. 244]

'width'
: The 'width' property gives the minimum width for the column.

'visibility'
: If the 'visibility' of a column is set to 'collapse', none of the cells in the column are rendered, and cells that span into other columns are clipped. In addition, the width of the table is diminished by the width the column would have taken up. See "Dynamic effects" [p. 251] below. Other values for 'visibility' have no effect.

Example(s):

Here are some examples of style rules that set properties on columns. The first two rules together implement the "rules" attribute of HTML 4.0 with a value of "cols". The third rule makes the "totals" column blue, the final two rules shows how to make a column a fixed size, by using the fixed layout algorithm [p. 247] .

```
col   { border-style: none solid }
table { border-style: hidden }
col.totals { background: blue }
table { table-layout: fixed }
col.totals { width: 5em }
```

17.4 Tables in the visual formatting model

In terms of the visual formatting model [p. 107] , a table may behave like a block-level [p. 109] or inline-level [p. 111] element. Tables have content, padding, borders, and margins.

In both cases, the table element generates an anonymous [p. 111] box that contains the table box itself and the caption's box (if present). The table and caption boxes retain their own content, padding, margin, and border areas, and the dimensions of the rectangular anonymous box are the smallest required to contain both. Vertical margins collapse where the table box and caption box touch. Any repositioning of the table must move the entire anonymous box, not just the table box, so that the caption follows the table.

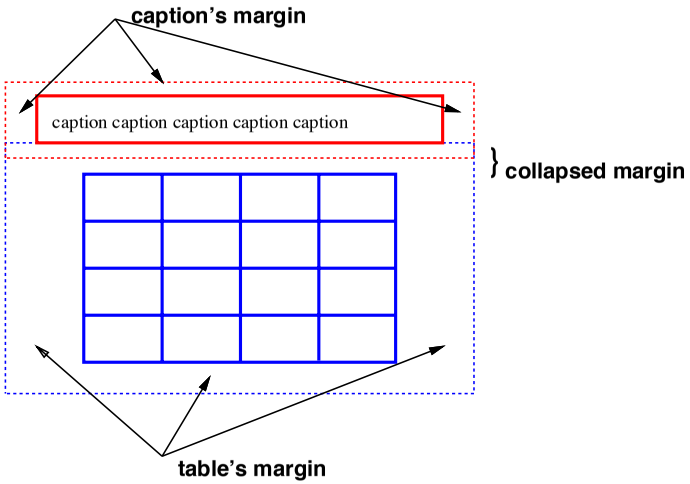


Diagram of a table with a caption above it; the bottom margin of the caption is collapsed with the top margin of the table.

17.4.1 Caption position and alignment

'caption-side'

Value:	top I bottom I inherit
Initial:	top
Applies to:	'table-caption' elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property specifies the position of the caption box with respect to the table box. Values have the following meanings:

top
Positions the caption box above the table box.

bottom
Positions the caption box below the table box.

Captions above or below a 'table' element are formatted very much as if they were a block element before or after the table, except that (1) they inherit inheritable properties from the table, and (2) they are not considered to be a block box for the purposes of any 'run-in' element that may precede the table.

A caption that is above or below a table box also behaves like a block box for width and height calculations; the width and height are calculated with respect to the table box's containing block.

To align caption content horizontally within the caption box, use the 'text-align' property.

Example(s):

In this example, the 'caption-side' property places captions below tables. The caption will be as wide as the parent of the table, and caption text will be left-justified.

```
caption { caption-side: bottom;
          width: auto;
          text-align: left }
```

17.5 Visual layout of table contents

Internal table elements generate rectangular boxes [p. 91] with content and borders. Cells have padding as well. Internal table elements do not have margins.

The visual layout of these boxes is governed by a rectangular, irregular grid of rows and columns. Each box occupies a whole number of grid cells, determined according to the following rules. These rules do not apply to HTML 4 or earlier HTML versions; HTML imposes its own limitations on row and column spans.

1. Each row box occupies one row of grid cells. Together, the row boxes fill the table from top to bottom in the order they occur in the source document (i.e., the table occupies exactly as many grid rows as there are row elements).
2. A row group occupies the same grid cells as the rows it contains.
3. A column box occupies one or more columns of grid cells. Column boxes are placed next to each other in the order they occur. The first column box may be either on the left or on the right, depending on the value of the 'direction' property of the table.
4. A column group box occupies the same grid cells as the columns it contains.
5. Cells may span several rows or columns. (Although CSS 2.1 doesn't define how the number of spanned rows or columns is determined, a user agent may have special knowledge about the source document; a future version of CSS may provide a way to express this knowledge in CSS syntax.) Each cell is thus a rectangular box, one or more grid cells wide and high. The top row of this rectangle is in the row specified by the cell's parent. The rectangle must be as far to the left as possible, but it may not overlap with any other cell box, and must be to the right of all cells in the same row that are earlier in the source document. (This constraint holds if the 'direction' property of the table is 'ltr'; if the 'direction' is 'rtl', interchange "left" and "right" in the previous sentence.)
6. A cell box cannot extend beyond the last row box of a table or row-group; the user agents must shorten it until it fits.

Note. *Table cells may be positioned, but this is not recommended: absolute and fixed positioning, as well as floating, remove a box from the flow, affecting table size.*

Here are two examples. The first is assumed to occur in an HTML document, the second an XHTML document:

```
<TABLE>
<TR><TD>1 <TD rowspan="2">2 <TD>3 <TD>4
<TR><TD colspan="2">5
</TABLE>
```

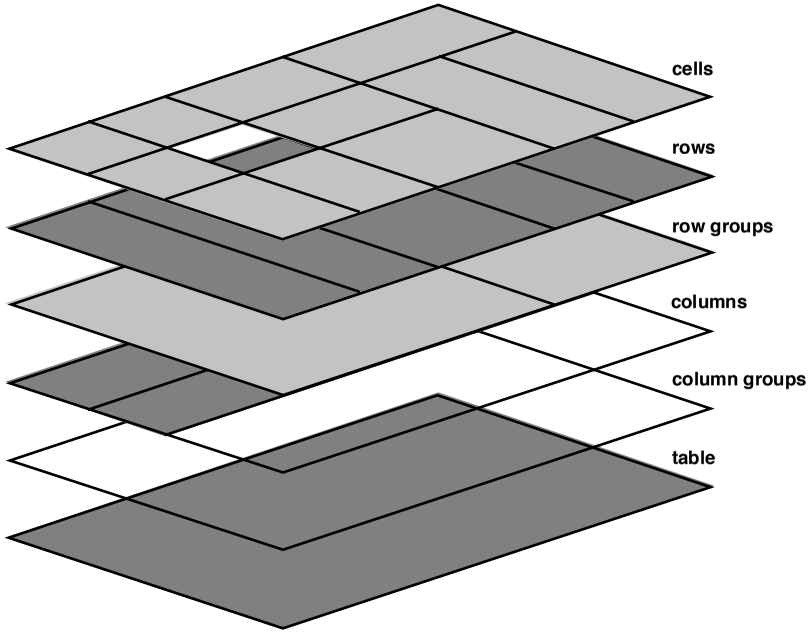
```
<table>
<tr><td>1 </td><td rowspan="2">2 </td><td>3 </td><td>4 <td><tr>
<tr><td colspan="2">5 <td><tr>
</table>
```

The second is formatted as in the figure on the right. However, the HTML table's rendering is explicitly undefined by HTML, and CSS doesn't try to define it. User agents are free to render it, e.g., as in the figure on the left.

On the left, one possible rendering of an erroneous HTML 4 table; on the right, the only possible formatting of a similar XHTML table.

17.5.1 Table layers and transparency

For the purposes of finding the background of each table cell, the different table elements may be thought of as being on six superimposed layers. The background set on an element in one of the layers will only be visible if the layers above it have a transparent background.



Schema of table layers.

1. The lowest layer is a single plane, representing the table box itself. Like all boxes, it may be transparent.
2. The next layer contains the column groups. Each column group extends from the top of the cells in the top row to the bottom of the cells on the bottom row and from the left edge of its leftmost column to the right edge of its rightmost column. The background extends to cover the full area of all cells that originate in the column group, but this extension does not affect background image positioning.

3. On top of the column groups are the areas representing the column boxes. Each column is as tall as the column groups and as wide as a normal (single-column-spanning) cell in the column. The background extends to cover the full area of all cells that originate in the column, even if they span outside the column, but this extension does not affect background image positioning.
4. Next is the layer containing the row groups. Each row group extends from the top left corner of its topmost cell in the first column to the bottom right corner of its bottommost cell in the last column.
5. The next to last layer contains the rows. Each row is as wide as the row groups and as tall as a normal (single-row-spanning) cell in the row. As with columns, the background extends to cover the full area of all cells that originate in the row, even if they span outside the row, but this this extension does not affect background image positioning.
6. The topmost layer contains the cells themselves. As the figure shows, although all rows contain the same number of cells, not every cell may have specified content. If the value of their 'empty-cells' property is 'hide' these "empty" cells are transparent through the cell, row, row group, column and column group backgrounds, letting the table background show through.

The edges of the rows, columns, row groups and column groups in the collapsing borders model [p. 254] coincide with the hypothetical grid lines on which the borders of the cells are centered. (And thus, in this model, the rows together exactly cover the table, leaving no gaps; ditto for the columns.) In the separated borders model, [p. 251] the edges coincide with the border edges [p. 92] of cells. (And thus, in this model, there may be gaps between the rows, columns, row groups or column groups, corresponding to the 'border-spacing' property.)

In the following example, the first row contains four cells, but the second row contains no cells, and thus the table background shines through, except where a cell from the first row spans into this row. The following HTML code and style rules

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD>
    <TITLE>Table example</TITLE>
    <STYLE type="text/css">
      TABLE { background: #ff0; border-collapse: collapse;
              empty-cells: hide }
      TD    { background: red; border: double black }
    </STYLE>
  </HEAD>
  <BODY>
    <TABLE>
      <TR>
        <TD> 1
        <TD rowspan="2"> 2
        <TD> 3
        <TD> 4
      </TR>
      <TR><TD></TD></TR>
    </TABLE>
  </BODY>
</HTML>
```

might be formatted as follows:

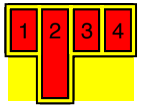


Table with three empty cells in the bottom row.

Note that if the table has 'border-collapse: separate', the background of the area given by the 'border-spacing' property is always the background of the table element. See the separated borders model [p. 251] .

17.5.2 Table width algorithms: the 'table-layout' property

CSS does not define an "optimal" layout for tables since, in many cases, what is optimal is a matter of taste. CSS does define constraints that user agents must respect when laying out a table. User agents may use any algorithm they wish to do so, and are free to prefer rendering speed over precision, except when the "fixed layout algorithm" is selected.

Note that this section overrides the rules that apply to calculating widths as described in section 10.3 [p. 153] . In particular, if the margins of a table are set to '0' and the width to 'auto', the table will not automatically size to fill its containing block. However, once the calculated value of 'width' for the table is found (using the algorithms given below or, when appropriate, some other UA dependant algorithm) then the other parts of section 10.3 do apply. Therefore a table *can* be centered using left and right 'auto' margins, for instance.

Future versions of CSS may introduce ways of making tables automatically fit their containing blocks.

'table-layout'

Value:	auto | fixed | inherit
Initial:	auto
Applies to:	'table' and 'inline-table' elements
Inherited:	no
Percentages:	N/A
Media:	visual
Computed value:	as specified

The 'table-layout' property controls the algorithm used to lay out the table cells, rows, and columns. Values have the following meaning:

fixed
Use the fixed table layout algorithm
auto
Use any automatic table layout algorithm

The two algorithms are described below.

Fixed table layout

With this (fast) algorithm, the horizontal layout of the table does not depend on the contents of the cells; it only depends on the table's width, the width of the columns, and borders or cell spacing.

The table's width may be specified explicitly with the 'width' property. A value of 'auto' (for both 'display: table' and 'display: inline-table') means use the automatic table layout [p. 248] algorithm. However, if the table is a block-level table ('display: table') in normal flow, a UA may (but does not have to) use the algorithm of 10.3.3 [p. 153] to compute a width and apply fixed table layout even if the specified width is 'auto'.

Example(s):

If a UA supports fixed table layout when 'width' is 'auto', the following will create a table that is 4em narrower than its containing block:

```
table { table-layout: fixed;
        margin-left: 2em;
        margin-right: 2em }
```

In the fixed table layout algorithm, the width of each column is determined as follows:

1. A column element with a value other than 'auto' for the 'width' property sets the width for that column.
2. Otherwise, a cell in the first row with a value other than 'auto' for the 'width' property sets the width for that column. If the cell spans more than one column, the width is divided over the columns.
3. Any remaining columns equally divide the remaining horizontal table space (minus borders or cell spacing).

The width of the table is then the greater of the value of the 'width' property for the table element and the sum of the column widths (plus cell spacing or borders). If the table is wider than the columns, the extra space should be distributed over the columns.

It is undefined what happens if a subsequent row has more columns than the first. When using 'table-layout: fixed', authors should not omit columns from the first row.

In this manner, the user agent can begin to lay out the table once the entire first row has been received. Cells in subsequent rows do not affect column widths. Any cell that has content that overflows uses the 'overflow' property to determine whether

to clip the overflow content.

Automatic table layout

In this algorithm (which generally requires no more than two passes), the table's width is given by the width of its columns (and intervening borders [p. 251]). This algorithm reflects the behavior of several popular HTML user agents at the writing of this specification. UAs are not required to implement this algorithm to determine the table layout in the case that 'table-layout' is 'auto'; they can use any other algorithm.

This algorithm may be inefficient since it requires the user agent to have access to all the content in the table before determining the final layout and may demand more than one pass.

Column widths are determined as follows:

1. Calculate the minimum content width (MCW) of each cell: the formatted content may span any number of lines but may not overflow the cell box. If the specified 'width' (W) of the cell is greater than MCW, W is the minimum cell width. A value of 'auto' means that MCW is the minimum cell width.

   Also, calculate the "maximum" cell width of each cell: formatting the content without breaking lines other than where explicit line breaks occur.
2. For each column, determine a maximum and minimum column width from the cells that span only that column. The minimum is that required by the cell with the largest minimum cell width (or the column 'width', whichever is larger). The maximum is that required by the cell with the largest maximum cell width (or the column 'width', whichever is larger).
3. For each cell that spans more than one column, increase the minimum widths of the columns it spans so that together, they are at least as wide as the cell. Do the same for the maximum widths. If possible, widen all spanned columns by approximately the same amount.

This gives a maximum and minimum width for each column. Column widths influence the final table width as follows:

1. If the 'table' or 'inline-table' element's 'width' property has a computed value (W) other than 'auto', the property's value as used for layout is the greater of W and the minimum width required by all the columns plus cell spacing or borders (MIN). If W is greater than MIN, the extra width should be distributed over the columns.
2. If the 'table' or 'inline-table' element has 'width: auto', the table width used for layout is the greater of the table's containing block width and MIN. However, if the maximum width required by the columns plus cell spacing or borders (MAX) is less than that of the containing block, use MAX.

A percentage value for a column width is relative to the table width. If the table has 'width: auto', a percentage represents a constraint on the column's width, which a UA should try to satisfy. (Obviously, this is not always possible: if the column's width

is ’110%’, the constraint cannot be satisfied.)

Note. *In this algorithm, rows (and row groups) and columns (and column groups) both constrain and are constrained by the dimensions of the cells they contain. Setting the width of a column may indirectly influence the height of a row, and vice versa.*

17.5.3 Table height algorithms

The height of a table is given by the ’height’ property for the ’table’ or ’inline-table’ element. A value of ’auto’ means that the height is the sum of the row heights plus any cell spacing or borders. Any other value specifies the height explicitly; the table may thus be taller or shorter than the height of its rows. CSS 2.1 does not specify rendering when the specified table height differs from the content height, in particular whether content height should override specified height; if it doesn’t, how extra space should be distributed among rows that add up to less than the specified table height; or, if the content height exceeds the specified table height, whether the UA should provide a scrolling mechanism. **Note.** Future versions of CSS may specify this further.

The height of a ’table-row’ element’s box is calculated once the user agent has all the cells in the row available: it is the maximum of the row’s specified ’height’ and the minimum height (MIN) required by the cells. A ’height’ value of ’auto’ for a ’table-row’ means the row height used for layout is MIN. MIN depends on cell box heights and cell box alignment (much like the calculation of a line box [p. 164] height). CSS 2.1 does not define what percentage values of ’height’ refer to when specified for table rows and row groups.

In CSS 2.1, the height of a cell box is the maximum of the table cell’s ’height’ property and the minimum height required by the content (MIN). A value of ’auto’ for ’height’ implies a that the value MIN will be used for layout. CSS 2.1 does not define what percentage values of ’height’ refer to when specified for table cells.

CSS 2.1 does not specify how cells that span more than row affect row height calculations except that the sum of the row heights involved must be great enough to encompass the cell spanning the rows.

The ’vertical-align’ property of each table cell determines its alignment within the row. Each cell’s content has a baseline, a top, a middle, and a bottom, as does the row itself. In the context of tables, values for ’vertical-align’ have the following meanings:

baseline
The baseline of the cell is put at the same height as the baseline of the first of the rows it spans (see below for the definition of baselines of cells and rows).

top
The top of the cell box is aligned with the top of the first row it spans.

bottom
The bottom of the cell box is aligned with the bottom of the last row it spans.

middle
The center of the cell is aligned with the center of the rows it spans.

sub, super, text-top, text-bottom
These values do not apply to cells; the cell is aligned at the baseline instead.

The baseline of a cell is the baseline of the first line box [p. 118] in the cell. If there is no text, the baseline is the baseline of whatever object is displayed in the cell, or, if it has none, the bottom of the cell box. The maximum distance between the top of the cell box and the baseline over all cells that have ’vertical-align: baseline’ is used to set the baseline of the row. Here is an example:

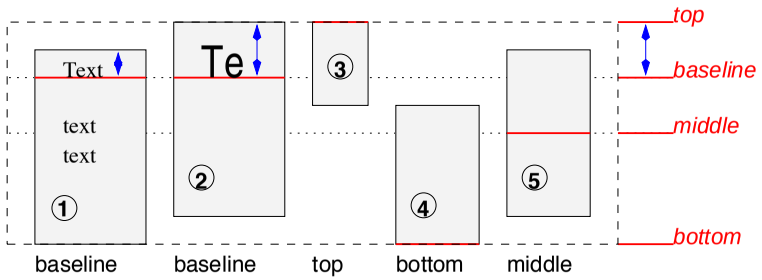


Diagram showing the effect of various values of ’vertical-align’ on table cells.

Cell boxes 1 and 2 are aligned at their baselines. Cell box 2 has the largest height above the baseline, so that determines the baseline of the row. Note that if there is no cell box aligned at its baseline, the row will not have (nor need) a baseline.

To avoid ambiguous situations, the alignment of cells proceeds in the following order:

1. First the cells that are aligned on their baseline are positioned. This will establish the baseline of the row. Next the cells with ’vertical-align: top’ are positioned.
2. The row now has a top, possibly a baseline, and a provisional height, which is the distance from the top to the lowest bottom of the cells positioned so far. (See conditions on the cell padding below.)
3. If any of the remaining cells, those aligned at the bottom or the middle, have a height that is larger than the current height of the row, the height of the row will be increased to the maximum of those cells, by lowering the bottom.
4. Finally the remaining cells are positioned.

Cell boxes that are smaller than the height of the row receive extra top or bottom padding.

17.5.4 Horizontal alignment in a column

The horizontal alignment of a cell's content within a cell box is specified with the 'text-align' property.

17.5.5 Dynamic row and column effects

The 'visibility' property takes the value 'collapse' for row, row group, column, and column group elements. This value causes the entire row or column to be removed from the display, and the space normally taken up by the row or column to be made available for other content. Contents of spanned rows and columns that intersect the collapsed column or row are clipped. The suppression of the row or column, however, does not otherwise affect the layout of the table. This allows dynamic effects to remove table rows or columns without forcing a re-layout of the table in order to account for the potential change in column constraints.

17.6 Borders

There are two distinct models for setting borders on table cells in CSS. One is most suitable for so-called separated borders around individual cells, the other is suitable for borders that are continuous from one end of the table to the other. Many border styles can be achieved with either model, so it is often a matter of taste which one is used.

'border-collapse'

Value:	collapse I separate I inherit
Initial:	separate
Applies to:	'table' and 'inline-table' elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

This property selects a table's border model. The value 'separate' selects the separated borders border model. The value 'collapse' selects the collapsing borders model. The models are described below.

17.6.1 The separated borders model

'border-spacing'

Value:	<length> <length>? I inherit
Initial:	0
Applies to:	'table' and 'inline-table' elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	two absolute lengths

The lengths specify the distance that separates adjacent cell borders. If one length is specified, it gives both the horizontal and vertical spacing. If two are specified, the first gives the horizontal spacing and the second the vertical spacing. Lengths may not be negative.

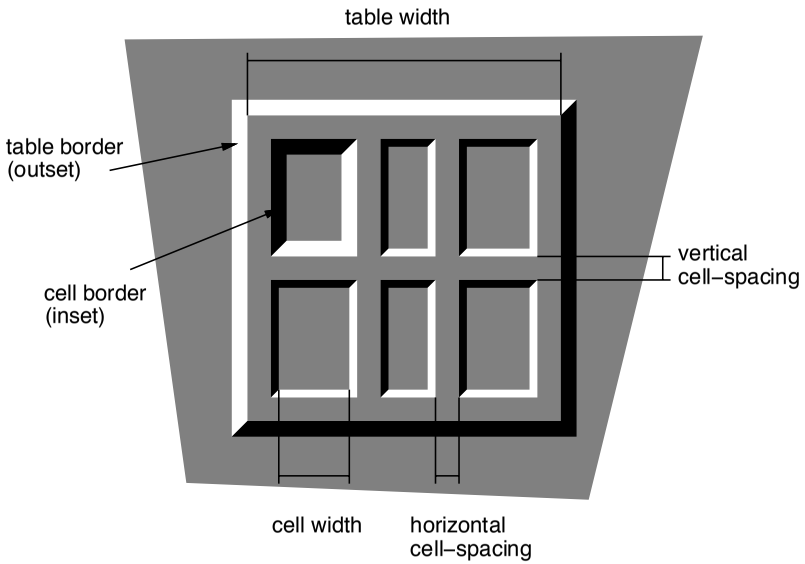
The distance between the table border and the borders of the cells on the edge of the table is the table's padding for that side, plus the relevant border spacing distance. For example, on the right hand side, the distance is *padding-right* + *horizontal border-spacing*.

In this model, each cell has an individual border. The 'border-spacing' property specifies the distance between the borders of adjacent cells. In this space, the row, column, row group, and column group backgrounds are invisible, allowing the table background to show through. Rows, columns, row groups, and column groups cannot have borders (i.e., user agents must ignore [p. 46] the border properties for those elements).

Example(s):

The table in the figure below could be the result of a style sheet like this:

```
table      { border: outset 10pt;
             border-collapse: separate;
             border-spacing: 15pt }
td         { border: inset 5pt }
td.special { border: inset 10pt }  /* The top-left cell */
```



A table with 'border-spacing' set to a length value. Note that each cell has its own border, and the table has a separate border as well.

Borders and Backgrounds around empty cells: the 'empty-cells' property

'empty-cells'

Value:	show I hide I inherit
Initial:	show
Applies to:	'table-cell' elements
Inherited:	yes
Percentages:	N/A
Media:	visual
Computed value:	as specified

In the separated borders model, this property controls the rendering of borders and backgrounds around cells that have no visible content. Empty cells and cells with the 'visibility' property set to 'hidden' are considered to have no visible content. Visible content includes " " and other whitespace except ASCII CR ("\0D"), LF ("\0A"), tab ("\09"), and space ("\20").

When this property has the value 'show', borders and backgrounds are drawn around/behind empty cells (like normal cells).

A value of 'hide' means that no borders or backgrounds are drawn around/behind empty cells (see point 6 in 17.5.1 [p. 244]). Furthermore, if all the cells in a row have a value of 'hide' and have no visible content, the entire row behaves as if it had 'display: none'.

Example(s):

The following rule causes borders and backgrounds to be drawn around all cells:

```
table { empty-cells: show }
```

17.6.2 The collapsing border model

In the collapsing border model, it is possible to specify borders that surround all or part of a cell, row, row group, column, and column group. Borders for HTML's "rule" attribute can be specified this way.

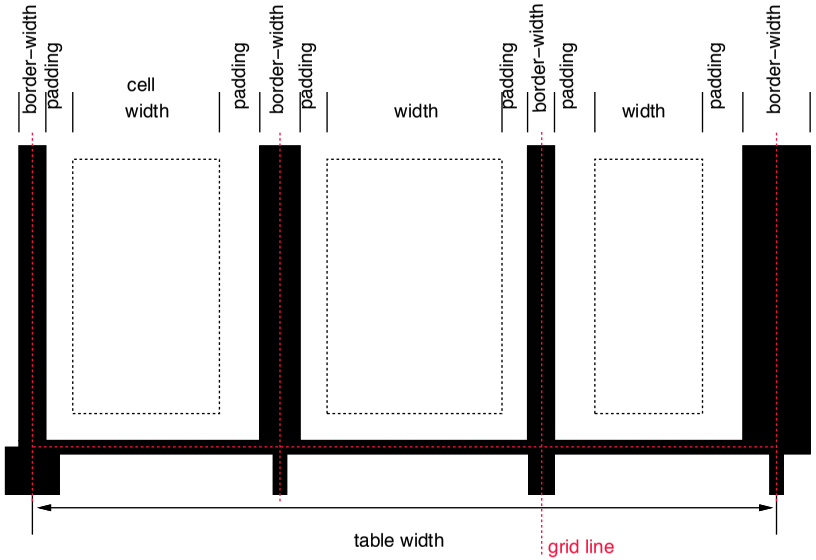
Borders are centered on the grid lines between the cells. User agents must find a consistent rule for rounding off in the case of an odd number of discrete units (screen pixels, printer dots).

The diagram below shows how the width of the table, the widths of the borders, the padding, and the cell width interact. Their relation is given by the following equation, which holds for every row of the table:

$$\textit{row-width} = (0.5 * \textit{border-width}_0) + \textit{padding-left}_1 + \textit{width}_1 + \textit{padding-right}_1 + \textit{border-width}_1 + \textit{padding-left}_2 + ... + \textit{padding-right}_n + (0.5 * \textit{border-width}_n)$$

Here n is the number of cells in the row, and $\textit{border-width}_i$ refers to the border between cells i and $i + 1$. Note only half of the two exterior borders are counted in the table width; the other half of these two borders lies in the margin area. Padding-left_i and padding-right_i refer to the left (resp., right) padding of cell i.

The half of the border that goes into the margin is taken into account when determining if the table overflows some ancestor (see 'overflow').



Schema showing the widths of cells and borders and the padding of cells.

Note that in this model, the width of the table includes half the table border. Also, in this model, a table doesn't have padding (but does have margins).

Border conflict resolution

In the collapsing border model, borders at every edge of every cell may be specified by border properties on a variety of elements that meet at that edge (cells, rows, row groups, columns, column groups, and the table itself), and these borders may vary in width, style, and color. The rule of thumb is that at each edge the most "eye catching" border style is chosen, except that any occurrence of the style 'hidden' unconditionally turns the border off.

The following rules determine which border style "wins" in case of a conflict:

1. Borders with the 'border-style' of 'hidden' take precedence over all other conflicting borders. Any border with this value suppresses all borders at this location.
2. Borders with a style of 'none' have the lowest priority. Only if the border properties of all the elements meeting at this edge are 'none' will the border be omitted (but note that 'none' is the default value for the border style.)
3. If none of the styles are 'hidden' and at least one of them is not 'none', then narrow borders are discarded in favor of wider ones. If several have the same 'border-width' then styles are preferred in this order: 'double', 'solid', 'dashed', 'dotted', 'ridge', 'outset', 'groove', and the lowest: 'inset'.
4. If border styles differ only in color, then a style set on a cell wins over one on a

row, which wins over a row group, column, column group and, lastly, table. It is undefined which color is used when two elements of the same type disagree.

Example(s):

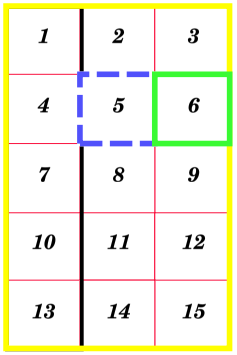
The following example illustrates the application of these precedence rules. This style sheet:

```
table          { border-collapse: collapse;
                 border: 5px solid yellow; }
*#col1         { border: 3px solid black; }
td             { border: 1px solid red; padding: 1em; }
td.solid-blue  { border: 5px dashed blue; }
td.solid-green { border: 5px solid green; }
```

with this HTML source:

```
<P>
<TABLE>
<COL id="col1"><COL id="col2"><COL id="col3">
<TR id="row1">
    <TD> 1
    <TD> 2
    <TD> 3
</TR>
<TR id="row2">
    <TD> 4
    <TD class="solid-blue"> 5
    <TD class="solid-green"> 6
</TR>
<TR id="row3">
    <TD> 7
    <TD> 8
    <TD> 9
</TR>
<TR id="row4">
    <TD> 10
    <TD> 11
    <TD> 12
</TR>
<TR id="row5">
    <TD> 13
    <TD> 14
    <TD> 15
</TR>
</TABLE>
```

would produce something like this:



An example of a table with collapsed borders.

Example(s):

The next example shows a table with horizontal rules between the rows. The top border of the table is set to 'hidden' to suppress the top border of the first row. This implements the "rules" attribute of HTML 4.0 (rules="rows").

```
table[rules=rows] tr { border-top: solid }
table[rules=rows]    { border-collapse: collapse;
                       border-top: hidden }
```

a	b	c
3	4	5
5	12	13

Table with horizontal rules between the rows.

In this case the same effect can also be achieved without setting a 'hidden' border on TABLE, by addressing the first row separately. Which method is preferred is a matter of taste.

```
tr:first-child { border-top: none }
tr { border-top: solid }
```

Example(s):

Here is another example of hidden collapsing borders:

Table with two omitted internal borders.

HTML source:

```
<TABLE style="border-collapse: collapse; border: solid;">
<TR><TD style="border-right: hidden; border-bottom: hidden">foo</TD>
    <TD style="border: solid">bar</TD></TR>
<TR><TD style="border: none">foo</TD>
    <TD style="border: solid">bar</TD></TR>
</TABLE>
```

17.6.3 Border styles

Some of the values of the 'border-style' have different meanings in tables than for other elements. In the list below they are marked with an asterisk.

none
: No border.

***hidden**
: Same as 'none', but in the collapsing border model [p. 254] , also inhibits any other border (see the section on border conflicts [p. 255]).

dotted
: The border is a series of dots.

dashed
: The border is a series of short line segments.

solid
: The border is a single line segment.

double
: The border is two solid lines. The sum of the two lines and the space between them equals the value of 'border-width'.

groove
: The border looks as though it were carved into the canvas.

ridge
: The opposite of 'grove': the border looks as though it were coming out of the canvas.

***inset**
: In the separated borders model [p. 251] , the border makes the entire box look as though it were embedded in the canvas. In the collapsing border model [p. 254] , same as 'ridge'.

***outset**
: In the separated borders model [p. 251] , the border makes the entire box look as though it were coming out of the canvas. In the collapsing border model [p. 254] , same as 'groove'.

18 User interface

Contents

18.1 Cursors: the ’cursor’ property

’cursor’

Value:	[[<uri> ,]* [auto | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help | progress]] | inherit
Initial:	auto
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	visual, interactive
Computed value:	absolute URI; otherwise as specified

This property specifies the type of cursor to be displayed for the pointing device. Values have the following meanings:

auto
: The UA determines the cursor to display based on the current context.

crosshair
: A simple crosshair (e.g., short line segments resembling a "+" sign).

default
: The platform-dependent default cursor. Often rendered as an arrow.

pointer
: The cursor is a pointer that indicates a link.

move
: Indicates something is to be moved.

e-resize, ne-resize, nw-resize, n-resize, se-resize, sw-resize, s-resize, w-resize
: Indicate that some edge is to be moved. For example, the ’se-resize’ cursor is used when the movement starts from the south-east corner of the box.

text
: Indicates text that may be selected. Often rendered as an I-beam.

wait,
: Indicates that the program is busy and the user should wait. Often rendered as a watch or hourglass.

progress
: A progress indicator. The program is performing some processing, but is different from ’wait’ in that the user may still interact with the program. Often rendered as a spinning beach ball, or an arrow with a watch or hourglass.

help
: Help is available for the object under the cursor. Often rendered as a question mark or a balloon.

<uri>
: The user agent retrieves the cursor from the resource designated by the URI. If the user agent cannot handle the first cursor of a list of cursors, it should attempt to handle the second, etc. If the user agent cannot handle any user-defined cursor, it must use the generic cursor at the end of the list.

Example(s):

```
:link,:visited { cursor: url(example.svg#linkcursor), url(hyper.cur), pointer }
```

This example sets the cursor on all hyperlinks (whether visited or not) to an external SVG cursor [p. ??] . User agents that don’t support SVG cursors would simply skip to the next value and attempt to use the "hyper.cur" cursor. If that cursor format was also not supported, the UA would skip to the next value and simply render the ’pointer’ cursor.

18.2 CSS2 System Colors

Note. *The CSS2 System Colors are deprecated in the CSS3 Color Module [p. ??] .*

In addition to being able to assign pre-defined color values [p. 53] to text, backgrounds, etc., CSS2 introduced a set of named color values that allows authors to specify colors in a manner that integrates them into the operating system’s graphic environment.

For systems that do not have a corresponding value, the specified value should be mapped to the nearest system value, or to a default color.

The following lists additional values for color-related CSS properties and their general meaning. Any color property (e.g., ’color’ or ’background-color’) can take one of the following names. Although these are case-insensitive, it is recommended that the mixed capitalization shown below be used, to make the names more legible.

ActiveBorder
: Active window border.

ActiveCaption
: Active window caption.

AppWorkspace
: Background color of multiple document interface.

Background
: Desktop background.

ButtonFace
: Face color for three-dimensional display elements.

ButtonHighlight
: Highlight color for three-dimensional display elements (for edges facing away from the light source).

ButtonShadow
: Shadow color for three-dimensional display elements.

ButtonText
: Text on push buttons.

CaptionText
: Text in caption, size box, and scrollbar arrow box.

GrayText
: Grayed (disabled) text. This color is set to #000 if the current display driver does not support a solid gray color.

Highlight
: Item(s) selected in a control.

HighlightText
: Text of item(s) selected in a control.

InactiveBorder
: Inactive window border.

InactiveCaption
: Inactive window caption.

InactiveCaptionText
: Color of text in an inactive caption.

InfoBackground
: Background color for tooltip controls.

InfoText
: Text color for tooltip controls.

Menu
: Menu background.

MenuText
: Text in menus.

Scrollbar
: Scroll bar gray area.

ThreeDDarkShadow
: Dark shadow for three-dimensional display elements.

ThreeDFace
: Face color for three-dimensional display elements.

ThreeDHighlight
: Highlight color for three-dimensional display elements.

ThreeDLightShadow
: Light color for three-dimensional display elements (for edges facing the light source).

ThreeDShadow
: Dark shadow for three-dimensional display elements.

Window
: Window background.

WindowFrame
: Window frame.

WindowText
: Text in windows.

Example(s):

For example, to set the foreground and background colors of a paragraph to the same foreground and background colors of the user's window, write the following:

```
p { color: WindowText; background-color: Window }
```

18.3 User preferences for fonts

As for colors, authors may specify fonts in a way that makes use of a user's system resources. Please consult the 'font' property for details.

18.4 Dynamic outlines: the 'outline' property

At times, style sheet authors may want to create outlines around visual objects such as buttons, active form fields, image maps, etc., to make them stand out. CSS 2.1 outlines differ from borders [p. 100] in the following ways:

1. Outlines do not take up space.
2. Outlines may be non-rectangular.

The outline properties control the style of these dynamic outlines.

'outline'

Value:	[<'outline-color'> || <'outline-style'> || <'outline-width'>] | inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	visual, interactive
Computed value:	see individual properties

'outline-width'

Value: <border-width> | inherit
Initial: medium
Applies to: all elements
Inherited: no
Percentages: N/A
Media: visual, interactive
Computed value: absolute length; '0' if the outline style is 'none' or 'hidden'

'outline-style'

Value: <border-style> | inherit
Initial: none
Applies to: all elements
Inherited: no
Percentages: N/A
Media: visual, interactive
Computed value: as specified

'outline-color'

Value: <color> | invert | inherit
Initial: invert
Applies to: all elements
Inherited: no
Percentages: N/A
Media: visual, interactive
Computed value: as specified

The outline created with the outline properties is drawn "over" a box, i.e., the outline is always on top, and doesn't influence the position or size of the box, or of any other boxes. Therefore, displaying or suppressing outlines does not cause reflow.

The outline may be drawn starting just outside the border edge [p. 92] .

Outlines may be non-rectangular. For example, if the element is broken across several lines, the outline is the minimum outline that encloses all the element's boxes. In contrast to borders [p. 100] , the outline is not open at the line box's end or start, but is always fully connected if possible.

The 'outline-width' property accepts the same values as 'border-width'.

The 'outline-style' property accepts the same values as 'border-style', except that 'hidden' is not a legal outline style.

The 'outline-color' accepts all colors, as well as the keyword 'invert'. 'Invert' is expected to perform a color inversion on the pixels on the screen. This is a common trick to ensure the focus border is visible, regardless of color background.

Conformant UAs may ignore the 'invert' value on platforms that do not support color inversion of the pixels on the screen. If the UA does not support the 'invert' value then the initial value of the 'outline-color' property is the value of the 'color' property, similar to the initial value of the 'border-top-color' property.

The 'outline' property is a shorthand property, and sets all three of 'outline-style', 'outline-width', and 'outline-color'.

Note. *The outline is the same on all sides. In contrast to borders, there is no 'outline-top' or 'outline-left' property.*

This specification does not define how multiple overlapping outlines are drawn, or how outlines are drawn for boxes that are partially obscured behind other elements.

Note. *Since the focus outline does not affect formatting (i.e., no space is left for it in the box model), it may well overlap other elements on the page.*

Example(s):

Here's an example of drawing a thick outline around a BUTTON element:

```
button { outline-width : thick }
```

Scripts may be used to dynamically change the width of the outline, without provoking a reflow.

18.4.1 Outlines and the focus

Graphical user interfaces may use outlines around elements to tell the user which element on the page has the *focus*. These outlines are in addition to any borders, and switching outlines on and off should not cause the document to reflow. The focus is the subject of user interaction in a document (e.g., for entering text, selecting a button, etc.). User agents supporting the interactive media group [p. 89] must keep track of where the focus lies and must also represent the focus. This may be done by using dynamic outlines in conjunction with the :focus pseudo-class.

Example(s):

For example, to draw a thick black line around an element when it has the focus, and a thick red line when it is active, the following rules can be used:

```
:focus  { outline: thick solid black }
:active { outline: thick solid red }
```

18.5 Magnification

The CSS working group considers that the magnification of a document or portions of a document should not be specified through style sheets. User agents may support such magnification in different ways (e.g., larger images, louder sounds, etc.)

When magnifying a page, UAs should preserve the relationships between positioned elements. For example, a comic strip may be composed of images with overlaid text elements. When magnifying this page, a user agent should keep the text within the comic strip balloon.

User interface

Appendix E. Elaborate description of Stacking Contexts

Contents

This chapter defines the CSS2.1 painting order in more detail than described in the rest of the specification.

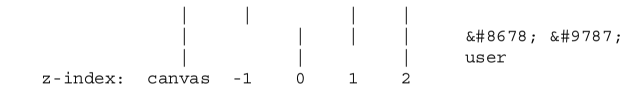
E.1 Definitions

Tree Order
: Preorder depth-first traversal of the *rendering* tree, in logical (not visual) order for bidirectional content, after taking into account properties that move boxes around such as the 'run-in' value of 'display'.

Element
: In this description, "element" refers to both actual elements and pseudo-elements. Pseudo-elements and anonymous boxes are treated as descendants in the appropriate places. For example, an outside list marker comes before an adjacent ':before' box in the line box, which comes before the content of the box, and so forth.

E.2 Painting order

The bottom of the stack is the furthest from the user, the top of the stack is the nearest to the user:



The stacking context background and most negative positioned stacking contexts are at the bottom of the stack, while the most positive positioned stacking contexts are at the top of the stack.

The canvas is transparent if contained within another, and given a UA-defined color if it is not. It is infinite in extent and contains the root element. Initially, the viewport is anchored with its top left corner at the canvas origin.

The stacking order for an element generating a stacking context (see the 'z-index' property) is:

1. If the element is a root element:
   1. background color of element over the entire canvas.
   2. background image of element, unclipped, painted at the origin that would be used if it was painted for the root element.
2. If the element is a block, list-item, or other block equivalent:
   1. background color of element unless it is the root element.
   2. background image of element unless it is the root element.
   3. border of element.

   Otherwise, if the element is a block level table:
   1. table backgrounds (color then image) unless it is the root element.
   2. column group backgrounds (color then image).
   3. column backgrounds (color then image).
   4. row group backgrounds (color then image).
   5. row backgrounds (color then image).
   6. cell backgrounds (color then image).
   7. all table borders (in tree order for separated borders).
3. Stacking contexts formed by positioned descendants with negative z-indices (excluding 0) in z-index order (most negative first) then tree order.
4. For all its in-flow, non-positioned, block-level descendants in tree order: If the element is a block, list-item, or other block equivalent:
   1. background color of element.
   2. background image of element.
   3. border of element.

   Otherwise, the element is a table:
   1. table backgrounds (color then image).
   2. column group backgrounds (color then image).
   3. column backgrounds (color then image).
   4. row group backgrounds (color then image).
   5. row backgrounds (color then image).
   6. cell backgrounds (color then image).
   7. all table borders (in tree order for separated borders).
5. All floating descendants, in tree order. For each one of these, treat the element as if it created a new stacking context, but any descendants which actually create a new stacking context should be considered part of the parent stacking context, not this new one.
6. First for the element, then for all its in-flow, non-positioned, block-level descendants in tree order:
   1. For each element in each line box of the element, if any, or if the element is inline-level (note that all elements in line boxes are forcably inline-level), on a per-line-box basis:
      1. Background color of element.
      2. Background image of element.
      3. Border of element.
      4. For inline elements:

1. Underline of element.
2. Overline of element.
3. Text of element.
4. Then all the element's in-flow, non-positioned, inline-level descendants, including anonymous inline elements generated for text nodes inside the element, in tree order:
    1. Jump to step 6.1 for this element.
5. Line-through of element.

For inline-block and inline-table elements:

1. For each one of these, treat the element as if it created a new stacking context, but any descendants which actually create a new stacking context should be considered part of the parent stacking context, not this new one.

For inline-level replaced elements:

1. The replaced content, atomically.

5. Optionally, the outline of the element (see 9 below).

2. Otherwise, if the element is a block-level replaced element, then the replaced content, atomically.
3. Optionally, if the element is block-level, the outline of the element (see 9 below).

7. All positioned descendants with 'z-index: auto' or 'z-index: 0', in tree order. For those with 'z-index: auto', treat the element as if it created a new stacking context, but any descendants which actually create a new stacking context should be considered part of the parent stacking context, not this new one. For those with 'z-index: 0', treat the stacking context generated atomically.
8. Stacking contexts formed by positioned descendants with z-indices greater than or equal to 1 in z-index order (smallest first) then tree order.
9. Finally, implementations that do not draw outlines in steps above must draw outlines from this stacking context at this stage.

E.3 Notes

If the root element is a block-level element, its background is only painted once, over the whole canvas. If the root element is not a block-level element, then its background is painted twice, once for canvas, and once for the box(es) generated by the element.

While the backgrounds of bidirectional inlines are painted in tree order, they are positioned in visual order. Since the positioning of inline backgrounds is unspecified in CSS2.1, the exact result of these two requirements is UA-defined. CSS3 may define this in more detail.

Elaborate description of Stacking Contexts

Appendix H: Has been intentionally left blank

Has been intentionally left blank

Appendix A. Aural style sheets

Contents

This chapter is informative. UAs are not required to implement the properties of this chapter in order to conform to CSS 2.1.

A.1 The media types ’aural’ and ’speech’

We expect that in a future level of CSS there will be new properties and values defined for speech output. Therefore CSS 2.1 reserves the ’speech’ media type (see chapter 7, "Media types" [p. 87]), but does not yet define which properties do or do not apply to it.

The properties in this appendix apply to a media type ’aural’, that was introduced in CSS2. The type ’aural’ is now deprecated.

This means that a style sheet such as

```
@media speech {
  body { voice-family: Paul }
}
```

is valid, but that its meaning is not defined by CSS 2.1, while


```
@media aural {
  body { voice-family: Paul }
}
```

is deprecated, but defined by this appendix.

A.2 Introduction to aural style sheets

The aural rendering of a document, already commonly used by the blind and print-impaired communities, combines speech synthesis and "auditory icons." Often such aural presentation occurs by converting the document to plain text and feeding this to a *screen reader* -- software or hardware that simply reads all the characters on the screen. This results in less effective presentation than would be the case if the document structure were retained. Style sheet properties for aural presentation may be used together with visual properties (mixed media) or as an aural alternative to visual presentation.

Besides the obvious accessibility advantages, there are other large markets for listening to information, including in-car use, industrial and medical documentation systems (intranets), home entertainment, and to help users learning to read or who have difficulty reading.

When using aural properties, the canvas consists of a three-dimensional physical space (sound surrounds) and a temporal space (one may specify sounds before, during, and after other sounds). The CSS properties also allow authors to vary the quality of synthesized speech (voice type, frequency, inflection, etc.).

Example(s):

```
h1, h2, h3, h4, h5, h6 {
    voice-family: paul;
    stress: 20;
    richness: 90;
    cue-before: url("ping.au")
}
p.heidi { azimuth: center-left }
p.peter { azimuth: right }
p.goat  { volume: x-soft }
```

This will direct the speech synthesizer to speak headers in a voice (a kind of "audio font") called "paul", on a flat tone, but in a very rich voice. Before speaking the headers, a sound sample will be played from the given URL. Paragraphs with class "heidi" will appear to come from front left (if the sound system is capable of spatial audio), and paragraphs of class "peter" from the right. Paragraphs with class "goat" will be very soft.

A.2.1 Angles

Angle values are denoted by <angle> in the text. Their format is a <number> immediately followed by an angle unit identifier.

Angle unit identifiers are:

- **deg**: degrees
- **grad**: grads
- **rad**: radians

Angle values may be negative. They should be normalized to the range 0-360deg by the user agent. For example, -10deg and 350deg are equivalent.

For example, a right angle is ’90deg’ or ’100grad’ or ’1.570796326794897rad’.

Like for <length>, the unit may be omitted, if the value is zero: ’0deg’ may be written as ’0’.

A.2.2 Times

Time values are denoted by <time> in the text. Their format is a <number> immediately followed by a time unit identifier.

Time unit identifiers are:

- **ms**: milliseconds
- **s**: seconds

Time values may not be negative.

Like for <length>, the unit may be omitted, if the value is zero: ’0s’ may be written as ’0’.

A.2.3 Frequencies

Frequency values are denoted by <frequency> in the text. Their format is a <number> immediately followed by a frequency unit identifier.

Frequency unit identifiers are:

- **Hz**: Hertz
- **kHz**: kilohertz

Frequency values may not be negative.

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

Like for <length>, the unit may be omitted, if the value is zero: ’0Hz’ may be written as ’0’.

A.3 Volume properties: ’volume’

’volume’

Value:	<number>	<percentage>	silent	x-soft	soft	medium	loud	x-loud	inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	refer to inherited value
Media:	aural
Computed value:	number

Volume refers to the median volume of the waveform. In other words, a highly inflected voice at a volume of 50 might peak well above that. The overall values are likely to be human adjustable for comfort, for example with a physical volume control (which would increase both the 0 and 100 values proportionately); what this property does is adjust the dynamic range.

Values have the following meanings:

<number>
: Any number between ’0’ and ’100’. ’0’ represents the *minimum audible* volume level and 100 corresponds to the *maximum comfortable* level.

<percentage>
: Percentage values are calculated relative to the inherited value, and are then clipped to the range ’0’ to ’100’.

silent
: No sound at all. The value ’0’ does not mean the same as ’silent’.

x-soft
: Same as ’0’.

soft
: Same as ’25’.

medium
: Same as ’50’.

loud
: Same as ’75’.

x-loud
: Same as ’100’.

User agents should allow the values corresponding to ’0’ and ’100’ to be set by the listener. No one setting is universally applicable; suitable values depend on the equipment in use (speakers, headphones), the environment (in car, home theater, library) and personal preferences. Some examples:

- A browser for in-car use has a setting for when there is lots of background noise. '0' would map to a fairly high level and '100' to a quite high level. The speech is easily audible over the road noise but the overall dynamic range is compressed. Cars with better insulation might allow a wider dynamic range.
- Another speech browser is being used in an apartment, late at night, or in a shared study room. '0' is set to a very quiet level and '100' to a fairly quiet level, too. As with the first example, there is a low slope; the dynamic range is reduced. The actual volumes are low here, whereas they were high in the first example.
- In a quiet and isolated house, an expensive hi-fi home theater setup. '0' is set fairly low and '100' to quite high; there is wide dynamic range.

The same author style sheet could be used in all cases, simply by mapping the '0' and '100' points suitably at the client side.

A.4 Speaking properties: 'speak'

'speak'

Value:	normal I none I spell-out I inherit
Initial:	normal
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

This property specifies whether text will be rendered aurally and if so, in what manner. The possible values are:

none
Suppresses aural rendering so that the element requires no time to render. Note, however, that descendants may override this value and will be spoken. (To be sure to suppress rendering of an element and its descendants, use the 'display' property).

normal
Uses language-dependent pronunciation rules for rendering an element and its children.

spell-out
Spells the text one letter at a time (useful for acronyms and abbreviations).

Note the difference between an element whose 'volume' property has a value of 'silent' and an element whose 'speak' property has the value 'none'. The former takes up the same time as if it had been spoken, including any pause before and after the element, but no sound is generated. The latter requires no time and is not rendered (though its descendants may be).

A.5 Pause properties: 'pause-before', 'pause-after', and 'pause'

'pause-before'

Value:	<time> I <percentage> I inherit
Initial:	0
Applies to:	all elements
Inherited:	no
Percentages:	see prose
Media:	aural
Computed value:	time

'pause-after'

Value:	<time> I <percentage> I inherit
Initial:	0
Applies to:	all elements
Inherited:	no
Percentages:	see prose
Media:	aural
Computed value:	time;;

These properties specify a pause to be observed before (or after) speaking an element's content. Values have the following meanings:

<time>
Expresses the pause in absolute time units (seconds and milliseconds).

<percentage>
Refers to the inverse of the value of the 'speech-rate' property. For example, if the speech-rate is 120 words per minute (i.e., a word takes half a second, or 500ms) then a 'pause-before' of 100% means a pause of 500 ms and a 'pause-before' of 20% means 100ms.

The pause is inserted between the element's content and any 'cue-before' or 'cue-after' content.

Authors should use relative units to create more robust style sheets in the face of large changes in speech-rate.

'pause'

Value:	[[<time> I <percentage>]{1,2}] I inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	see descriptions of 'pause-before' and 'pause-after'
Media:	aural
Computed value:	see individual properties

The 'pause' property is a shorthand for setting 'pause-before' and 'pause-after'. If two values are given, the first value is 'pause-before' and the second is 'pause-after'. If only one value is given, it applies to both properties.

Example(s):

```
h1 { pause: 20ms } /* pause-before: 20ms; pause-after: 20ms */
h2 { pause: 30ms 40ms } /* pause-before: 30ms; pause-after: 40ms */
h3 { pause-after: 10ms } /* pause-before unspecified; pause-after: 10ms */
```

A.6 Cue properties: 'cue-before', 'cue-after', and 'cue'

'cue-before'

Value:	<uri> I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural
Computed value:	absolute URI or 'none'

'cue-after'

Value:	<uri> I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural
Computed value:	absolute URI or 'none'

Auditory icons are another way to distinguish semantic elements. Sounds may be played before and/or after the element to delimit it. Values have the following meanings:

<uri>
The URI must designate an auditory icon resource. If the URI resolves to something other than an audio file, such as an image, the resource should be ignored

and the property treated as if it had the value 'none'.

none
No auditory icon is specified.

Example(s):

```
a {cue-before: url("bell.aiff"); cue-after: url("dong.wav") }
h1 {cue-before: url("pop.au"); cue-after: url("pop.au") }
```

'cue'

Value:	[<'cue-before'> II <'cue-after'>] I inherit
Initial:	see individual properties
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural
Computed value:	see individual properties

The 'cue' property is a shorthand for setting 'cue-before' and 'cue-after'. If two values are given, the first value is 'cue-before' and the second is 'cue-after'. If only one value is given, it applies to both properties.

Example(s):

The following two rules are equivalent:

```
h1 {cue-before: url("pop.au"); cue-after: url("pop.au") }
h1 {cue: url("pop.au") }
```

If a user agent cannot render an auditory icon (e.g., the user's environment does not permit it), we recommend that it produce an alternative cue.

Please see the sections on the :before and :after pseudo-elements [p. 177] for information on other content generation techniques. 'cue-before' sounds and 'pause-before' gaps are inserted before content from the ':before' pseudo-element. Similarly, 'pause-after' gaps and 'cue-after' sounds are inserted after content from the ':after' pseudo-element.

A.7 Mixing properties: 'play-during'

'play-during'

Value:	<uri> [mix || repeat]?	auto	none	inherit
Initial:	auto
Applies to:	all elements
Inherited:	no
Percentages:	N/A
Media:	aural
Computed value:	absolute URI, rest as specified

Similar to the 'cue-before' and 'cue-after' properties, this property specifies a sound to be played as a background while an element's content is spoken. Values have the following meanings:

<uri>
The sound designated by this <uri> is played as a background while the element's content is spoken.

mix
When present, this keyword means that the sound inherited from the parent element's 'play-during' property continues to play and the sound designated by the <uri> is mixed with it. If 'mix' is not specified, the element's background sound replaces the parent's.

repeat
When present, this keyword means that the sound will repeat if it is too short to fill the entire duration of the element. Otherwise, the sound plays once and then stops. This is similar to the 'background-repeat' property. If the sound is too long for the element, it is clipped once the element has been spoken.

auto
The sound of the parent element continues to play (it is not restarted, which would have been the case if this property had been inherited).

none
This keyword means that there is silence. The sound of the parent element (if any) is silent during the current element and continues after the current element.

Example(s):

```
blockquote.sad { play-during: url("violins.aiff") }
blockquote Q   { play-during: url("harp.wav") mix }
span.quiet     { play-during: none }
```

A.8 Spatial properties: 'azimuth' and 'elevation'

Spatial audio is an important stylistic property for aural presentation. It provides a natural way to tell several voices apart, as in real life (people rarely all stand in the same spot in a room). Stereo speakers produce a lateral sound stage. Binaural headphones or the increasingly popular 5-speaker home theater setups can generate full surround sound, and multi-speaker setups can create a true three-dimensional sound stage. VRML 2.0 also includes spatial audio, which implies that in time consumer-priced spatial audio hardware will become more widely available.

'azimuth'

Value:	<angle>	[[left-side	far-left	left	center-left	center	center-right	right	far-right	right-side] || behind]	leftwards	rightwards	inherit
Initial:	center
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	normalized angle

Values have the following meanings:

<angle>
Position is described in terms of an angle within the range '-360deg' to '360deg'. The value '0deg' means directly ahead in the center of the sound stage. '90deg' is to the right, '180deg' behind, and '270deg' (or, equivalently and more conveniently, '-90deg') to the left.

left-side
Same as '270deg'. With 'behind', '270deg'.

far-left
Same as '300deg'. With 'behind', '240deg'.

left
Same as '320deg'. With 'behind', '220deg'.

center-left
Same as '340deg'. With 'behind', '200deg'.

center
Same as '0deg'. With 'behind', '180deg'.

center-right
Same as '20deg'. With 'behind', '160deg'.

right
Same as '40deg'. With 'behind', '140deg'.

far-right
Same as '60deg'. With 'behind', '120deg'.

right-side
Same as '90deg'. With 'behind', '90deg'.

leftwards
Moves the sound to the left, relative to the current angle. More precisely, subtracts 20 degrees. Arithmetic is carried out modulo 360 degrees. Note that 'leftwards' is more accurately described as "turned counter-clockwise," since it *always* subtracts 20 degrees, even if the inherited azimuth is already behind the listener (in which case the sound actually appears to move to the right).

rightwards
Moves the sound to the right, relative to the current angle. More precisely, adds 20 degrees. See 'leftwards' for arithmetic.

This property is most likely to be implemented by mixing the same signal into different channels at differing volumes. It might also use phase shifting, digital delay, and other such techniques to provide the illusion of a sound stage. The precise means used to achieve this effect and the number of speakers used to do so are user agent-dependent; this property merely identifies the desired end result.

Example(s):

```
h1   { azimuth: 30deg }
td.a { azimuth: far-right }          /*  60deg */
#12  { azimuth: behind far-right }   /* 120deg */
p.comment { azimuth: behind }        /* 180deg */
```

If spatial-azimuth is specified and the output device cannot produce sounds *behind* the listening position, user agents should convert values in the rearwards hemisphere to forwards hemisphere values. One method is as follows:

- if 90deg < x <= 180deg then x := 180deg - x
- if 180deg < x <= 270deg then x := 540deg - x

'elevation'

Value:	<angle> I below I level I above I higher I lower I inherit
Initial:	level
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	normalized angle

Values of this property have the following meanings:

<angle>
Specifies the elevation as an angle, between '-90deg' and '90deg'. '0deg' means on the forward horizon, which loosely means level with the listener. '90deg' means directly overhead and '-90deg' means directly below.
below
Same as '-90deg'.
level
Same as '0deg'.
above
Same as '90deg'.
higher
Adds 10 degrees to the current elevation.
lower
Subtracts 10 degrees from the current elevation.

The precise means used to achieve this effect and the number of speakers used to do so are undefined. This property merely identifies the desired end result.

Example(s):

```
h1   { elevation: above }
tr.a { elevation: 60deg }
tr.b { elevation: 30deg }
tr.c { elevation: level }
```

A.9 Voice characteristic properties: 'speech-rate', 'voice-family', 'pitch', 'pitch-range', 'stress', and 'richness'

'speech-rate'

Value:	<number> I x-slow I slow I medium I fast I x-fast I faster I slower I inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	number

This property specifies the speaking rate. Note that both absolute and relative keyword values are allowed (compare with 'font-size'). Values have the following meanings:

<number>
Specifies the speaking rate in words per minute, a quantity that varies somewhat by language but is nevertheless widely supported by speech synthesizers.
x-slow
Same as 80 words per minute.
slow
Same as 120 words per minute
medium
Same as 180 - 200 words per minute.
fast
Same as 300 words per minute.
x-fast
Same as 500 words per minute.
faster
Adds 40 words per minute to the current speech rate.
slower
Subtracts 40 words per minutes from the current speech rate.

'voice-family'

Value:	[[<specific-voice>	<generic-voice>],]* [<specific-voice>	<generic-voice>]	inherit
Initial:	depends on user agent
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

The value is a comma-separated, prioritized list of voice family names (compare with 'font-family'). Values have the following meanings:

<generic-voice>
Values are voice families. Possible values are 'male', 'female', and 'child'.
<specific-voice>
Values are specific instances (e.g., comedian, trinoids, carlos, lani).

Example(s):

```
h1 { voice-family: announcer, male }
p.part.romeo  { voice-family: romeo, male }
p.part.juliet { voice-family: juliet, female }
```

Names of specific voices may be quoted, and indeed must be quoted if any of the words that make up the name does not conform to the syntax rules for identifiers [p. 37] . It is also recommended to quote specific voices with a name consisting of more than one word. If quoting is omitted, any whitespace [p. 40] characters before and after the voice family name are ignored and any sequence of whitespace characters inside the voice family name is converted to a single space.

'pitch'

Value:	<frequency>	x-low	low	medium	high	x-high	inherit
Initial:	medium
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	frequency

Specifies the average pitch (a frequency) of the speaking voice. The average pitch of a voice depends on the voice family. For example, the average pitch for a standard male voice is around 120Hz, but for a female voice, it's around 210Hz.

Values have the following meanings:

<frequency>
Specifies the average pitch of the speaking voice in hertz (Hz).
x-low, **low**, **medium**, **high**, **x-high**
These values do not map to absolute frequencies since these values depend on the voice family. User agents should map these values to appropriate frequencies based on the voice family and user environment. However, user agents must map these values in order (i.e., 'x-low' is a lower frequency than 'low', etc.).

'pitch-range'

Value:	<number>	inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

Specifies variation in average pitch. The perceived pitch of a human voice is determined by the fundamental frequency and typically has a value of 120Hz for a male voice and 210Hz for a female voice. Human languages are spoken with varying inflection and pitch; these variations convey additional meaning and emphasis. Thus, a highly animated voice, i.e., one that is heavily inflected, displays a high pitch range. This property specifies the range over which these variations occur, i.e., how much the fundamental frequency may deviate from the average pitch.

Values have the following meanings:

<number>
A value between '0' and '100'. A pitch range of '0' produces a flat, monotonic voice. A pitch range of 50 produces normal inflection. Pitch ranges greater than 50 produce animated voices.

'stress'

Value:	<number>	inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

Specifies the height of "local peaks" in the intonation contour of a voice. For example, English is a **stressed** language, and different parts of a sentence are assigned primary, secondary, or tertiary stress. The value of 'stress' controls the amount of inflection that results from these stress markers. This property is a companion to the 'pitch-range' property and is provided to allow developers to exploit higher-end auditory displays.

Values have the following meanings:

<number>
A value, between '0' and '100'. The meaning of values depends on the language being spoken. For example, a level of '50' for a standard, English-speaking male voice (average pitch = 122Hz), speaking with normal intonation and emphasis would have a different meaning than '50' for an Italian voice.

'richness'

Value:	<number> I inherit
Initial:	50
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

Specifies the richness, or brightness, of the speaking voice. A rich voice will "carry" in a large room, a smooth voice will not. (The term "smooth" refers to how the wave form looks when drawn.)

Values have the following meanings:

<number>
A value between '0' and '100'. The higher the value, the more the voice will carry. A lower value will produce a soft, mellifluous voice.

A.10 Speech properties: 'speak-punctuation' and 'speak-numeral'

An additional speech property, 'speak-header', is described below.

'speak-punctuation'

Value:	code I none I inherit
Initial:	none
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

This property specifies how punctuation is spoken. Values have the following meanings:

code
Punctuation such as semicolons, braces, and so on are to be spoken literally.
none
Punctuation is not to be spoken, but instead rendered naturally as various pauses.

'speak-numeral'

Value:	digits I continuous I inherit
Initial:	continuous
Applies to:	all elements
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

This property controls how numerals are spoken. Values have the following meanings:

digits
Speak the numeral as individual digits. Thus, "237" is spoken "Two Three Seven".
continuous
Speak the numeral as a full number. Thus, "237" is spoken "Two hundred thirty seven". Word representations are language-dependent.

A.11 Audio rendering of tables

When a table is spoken by a speech generator, the relation between the data cells and the header cells must be expressed in a different way than by horizontal and vertical alignment. Some speech browsers may allow a user to move around in the 2-dimensional space, thus giving them the opportunity to map out the spatially represented relations. When that is not possible, the style sheet must specify at which points the headers are spoken.

A.11.1 Speaking headers: the 'speak-header' property

'speak-header'

Value:	once I always I inherit
Initial:	once
Applies to:	elements that have table header information
Inherited:	yes
Percentages:	N/A
Media:	aural
Computed value:	as specified

This property specifies whether table headers are spoken before every cell, or only before a cell when that cell is associated with a different header than the previous cell. Values have the following meanings:

once
The header is spoken one time, before a series of cells.

always
The header is spoken before every pertinent cell.

Each document language may have different mechanisms that allow authors to specify headers. For example, in HTML 4.0 ([HTML40]), it is possible to specify header information with three different attributes ("headers", "scope", and "axis"), and the specification gives an algorithm for determining header information when these attributes have not been specified.

Travel Expense Report

	Meals	Hotels	Transport	subtotals
San Jose				
25-Aug-97	37.74	112.00	45.00	
26-Aug-97	27.28	112.00	45.00	
subtotals	65.02	224.00	90.00	379.02
Seattle				
27-Aug-97	96.25	109.00	36.00	
28-Aug-97	35.00	109.00	36.00	
subtotals	131.25	218.00	72.00	421.25
Totals	196.27	442.00	162.00	**800.27**

Image of a table with header cells ("San Jose" and "Seattle") that are not in the same column or row as the data they apply to.

This HTML example presents the money spent on meals, hotels and transport in two locations (San Jose and Seattle) for successive days. Conceptually, you can think of the table in terms of an n-dimensional space. The headers of this space are: location, day, category and subtotal. Some cells define marks along an axis while

others give money spent at points within this space. The markup for this table is:

```
<TABLE>
<CAPTION>Travel Expense Report</CAPTION>
<TR>
  <TH></TH>
  <TH>Meals</TH>
  <TH>Hotels</TH>
  <TH>Transport</TH>
  <TH>subtotal</TH>
</TR>
<TR>
  <TH id="san-jose" axis="san-jose">San Jose</TH>
</TR>
<TR>
  <TH headers="san-jose">25-Aug-97</TH>
  <TD>37.74</TD>
  <TD>112.00</TD>
  <TD>45.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="san-jose">26-Aug-97</TH>
  <TD>27.28</TD>
  <TD>112.00</TD>
  <TD>45.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="san-jose">subtotal</TH>
  <TD>65.02</TD>
  <TD>224.00</TD>
  <TD>90.00</TD>
  <TD>379.02</TD>
</TR>
<TR>
  <TH id="seattle" axis="seattle">Seattle</TH>
</TR>
<TR>
  <TH headers="seattle">27-Aug-97</TH>
  <TD>96.25</TD>
  <TD>109.00</TD>
  <TD>36.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="seattle">28-Aug-97</TH>
  <TD>35.00</TD>
  <TD>109.00</TD>
  <TD>36.00</TD>
  <TD></TD>
</TR>
<TR>
  <TH headers="seattle">subtotal</TH>
  <TD>131.25</TD>
  <TD>218.00</TD>
  <TD>72.00</TD>
```

```
  <TD>421.25</TD>
</TR>
<TR>
  <TH>Totals</TH>
  <TD>196.27</TD>
  <TD>442.00</TD>
  <TD>162.00</TD>
  <TD>800.27</TD>
</TR>
</TABLE>
```

By providing the data model in this way, authors make it possible for speech enabled-browsers to explore the table in rich ways, e.g., each cell could be spoken as a list, repeating the applicable headers before each data cell:

```
  San Jose, 25-Aug-97, Meals:  37.74
  San Jose, 25-Aug-97, Hotels:  112.00
  San Jose, 25-Aug-97, Transport:  45.00
 ...
```

The browser could also speak the headers only when they change:

```
San Jose, 25-Aug-97, Meals: 37.74
    Hotels: 112.00
    Transport: 45.00
  26-Aug-97, Meals: 27.28
    Hotels: 112.00
...
```

A.12 Sample style sheet for HTML

This style sheet describes a possible rendering of HTML 4.0:

```
@media aural {
h1, h2, h3,
h4, h5, h6    { voice-family: paul, male; stress: 20; richness: 90 }
h1            { pitch: x-low; pitch-range: 90 }
h2            { pitch: x-low; pitch-range: 80 }
h3            { pitch: low; pitch-range: 70 }
h4            { pitch: medium; pitch-range: 60 }
h5            { pitch: medium; pitch-range: 50 }
h6            { pitch: medium; pitch-range: 40 }
li, dt, dd    { pitch: medium; richness: 60 }
dt            { stress: 80 }
pre, code, tt { pitch: medium; pitch-range: 0; stress: 0; richness: 80 }
em            { pitch: medium; pitch-range: 60; stress: 60; richness: 50 }
strong        { pitch: medium; pitch-range: 60; stress: 90; richness: 90 }
dfn           { pitch: high; pitch-range: 60; stress: 60 }
s, strike     { richness: 0 }
i             { pitch: medium; pitch-range: 60; stress: 60; richness: 50 }
b             { pitch: medium; pitch-range: 60; stress: 90; richness: 90 }
u             { richness: 0 }
a:link        { voice-family: harry, male }
a:visited     { voice-family: betty, female }
a:active      { voice-family: betty, female; pitch-range: 80; pitch: x-high }
}
```


A.13 Emacspeak

For information, here is the list of properties implemented by Emacspeak, a speech subsystem for the Emacs editor.

- voice-family
- stress (but with a different range of values)
- richness (but with a different range of values)
- pitch (but with differently named values)
- pitch-range (but with a different range of values)

(We thank T. V. Raman for the information about implementation status of aural properties.)

Appendix D. Default style sheet for HTML 4.0

This appendix is informative, not normative.

This style sheet describes the typical formatting of all HTML 4.0 ([HTML40]) elements based on extensive research into current UA practice. Developers are encouraged to use it as a default style sheet in their implementations.

The full presentation of some HTML elements cannot be expressed in CSS 2.1, including replaced [p. 32] elements ("img", "object"), scripting elements ("script", "applet"), form control elements, and frame elements.

For other elements, the legacy presentation can be described in CSS but the solution removes the element. For example, the FONT element can be replaced by attaching CSS declarations to other elements (e.g., DIV). Likewise, legacy presentation of presentational attributes (e.g., the "border" attribute on TABLE) can be described in CSS, but the markup in the source document must be changed.

```
address,
blockquote,
body, dd, div,
dl, dt, fieldset, form,
frame, frameset,
h1, h2, h3, h4,
h5, h6, noframes,
ol, p, ul, center,
dir, hr, menu, pre   { display: block }
li              { display: list-item }
head            { display: none }
table           { display: table }
tr              { display: table-row }
thead           { display: table-header-group }
tbody           { display: table-row-group }
tfoot           { display: table-footer-group }
col             { display: table-column }
colgroup        { display: table-column-group }
td, th          { display: table-cell; }
caption         { display: table-caption }
th              { font-weight: bolder; text-align: center }
caption         { text-align: center }
body            { margin: 8px; line-height: 1.12 }
h1              { font-size: 2em; margin: .67em 0 }
h2              { font-size: 1.5em; margin: .75em 0 }
h3              { font-size: 1.17em; margin: .83em 0 }
h4, p,
blockquote, ul,
fieldset, form,
ol, dl, dir,
menu            { margin: 1.12em 0 }
h5              { font-size: .83em; margin: 1.5em 0 }
h6              { font-size: .75em; margin: 1.67em 0 }
h1, h2, h3, h4,
h5, h6, b,
```



```
strong          { font-weight: bolder }
blockquote      { margin-left: 40px; margin-right: 40px }
i, cite, em,
var, address    { font-style: italic }
pre, tt, code,
kbd, samp       { font-family: monospace }
pre             { white-space: pre }
button, textarea,
input, object,
select          { display:inline-block; }
big             { font-size: 1.17em }
small, sub, sup { font-size: .83em }
sub             { vertical-align: sub }
sup             { vertical-align: super }
thead, tbody,
tfoot           { vertical-align: middle }
td, th          { vertical-align: inherit }
s, strike, del  { text-decoration: line-through }
hr              { border: 1px inset }
ol, ul, dir,
menu, dd        { margin-left: 40px }
ol              { list-style-type: decimal }
ol ul, ul ol,
ul ul, ol ol    { margin-top: 0; margin-bottom: 0 }
u, ins          { text-decoration: underline }
br:before       { content: "\A" }
:before, :after { white-space: pre-line }
center          { text-align: center }
abbr, acronym   { font-variant: small-caps; letter-spacing: 0.1em }
:link, :visited { text-decoration: underline }
:focus          { outline: thin dotted invert }

/* Begin bidirectionality settings (do not change) */
BDO[DIR="ltr"]  { direction: ltr; unicode-bidi: bidi-override }
BDO[DIR="rtl"]  { direction: rtl; unicode-bidi: bidi-override }

*[DIR="ltr"]    { direction: ltr; unicode-bidi: embed }
*[DIR="rtl"]    { direction: rtl; unicode-bidi: embed }

@media print {
  h1            { page-break-before: always }
  h1, h2, h3,
  h4, h5, h6    { page-break-after: avoid }
  ul, ol, dl    { page-break-before: avoid }
}
```

XSL Transformations (XSLT) Version 1.0

W3C Recommendation 16 November 1999

This version:
http://www.w3.org/TR/1999/REC-xslt-19991116
(available in XML or HTML)
Latest version:
http://www.w3.org/TR/xslt
Previous versions:
http://www.w3.org/TR/1999/PR-xslt-19991008
http://www.w3.org/1999/08/WD-xslt-19990813
http://www.w3.org/1999/07/WD-xslt-19990709
http://www.w3.org/TR/1999/WD-xslt-19990421
http://www.w3.org/TR/1998/WD-xsl-19981216
http://www.w3.org/TR/1998/WD-xsl-19980818
Editor:
James Clark <jjc@jclark.com>

Abstract

This specification defines the syntax and semantics of XSLT, which is a language for transforming XML documents into other XML documents.

XSLT is designed for use as part of XSL, which is a stylesheet language for XML. In addition to XSLT, XSL includes an XML vocabulary for specifying formatting. XSL specifies the styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL.

Status of this document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from other documents. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

The list of known errors in this specification is available at http://www.w3.org/1999/11/REC-xslt-19991116-errata.

Comments on this specification may be sent to xsl-editors@w3.org; archives of the comments are available. Public discussion of XSL, including XSL Transformations, takes place on the XSL-List mailing list.

The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/Style/XSL/translations.html.

A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

This specification has been produced as part of the W3C Style activity.

Table of contents

Appendices

1 Introduction

This specification defines the syntax and semantics of the XSLT language. A transformation in the XSLT language is expressed as a well-formed XML document [XML] conforming to the Namespaces in XML Recommendation [XML Names], which may include both elements that are defined by XSLT and elements that are not defined by XSLT. XSLT-defined elements are distinguished by belonging to a specific XML namespace (see [**2.1 XSLT Namespace**]), which is referred to in this specification as the **XSLT namespace**. Thus this specification is a definition of the syntax and semantics of the XSLT namespace.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree. The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create part of the result tree. The result tree is separate from the source tree. The structure of the result tree can be completely different from the structure of the source tree. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

A transformation expressed in XSLT is called a stylesheet. This is because, in the case when XSLT is transforming into the XSL formatting vocabulary, the transformation functions as a stylesheet.

This document does not specify how an XSLT stylesheet is associated with an XML document. It is recommended that XSL processors support the mechanism described in [XML Stylesheet]. When this or any other mechanism yields a sequence of more than one XSLT stylesheet to be applied simultaneously to a XML document, then the effect should be the same as applying a single stylesheet that imports each member of the sequence in order (see [**2.6.2 Stylesheet Import**]).

A stylesheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

A template is instantiated for a particular source element to create part of the result tree. A template can contain elements that specify literal result element structure. A template can also contain elements from the XSLT namespace that are instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates. Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. Note that elements are only processed when they have been selected by the execution of an instruction. The result tree is constructed by finding the template rule for the root node and instantiating its template.

In the process of finding the applicable template rule, more than one template rule may have a pattern that matches a given element. However, only one template rule will be applied. The method for deciding which template rule to apply is described in [**5.5 Conflict Resolution for Template Rules**].

A single template by itself has considerable power: it can create structures of arbitrary complexity; it can pull string values out of arbitrary locations in the source tree; it can generate structures that are repeated according to the occurrence of elements in the source tree. For simple transformations where the structure of the result tree is independent of the structure of the source tree, a stylesheet can often consist of only a single template, which functions as a template for the complete result tree. Transformations on XML documents that represent data are often of this kind (see [**D.2 Data Example**]). XSLT allows a simplified syntax for such stylesheets (see [**2.3 Literal Result Element as Stylesheet**]).

When a template is instantiated, it is always instantiated with respect to a **current node** and a **current node list**. The current node is always a member of the current node list. Many operations in XSLT are relative to the current node. Only a few instructions change the current node list or the current node (see [**5 Template Rules**] and [**8 Repetition**]); during the instantiation of one of these instructions, the current node list changes to a new list of nodes and each member of this new list becomes the current node in turn; after the instantiation of the instruction is complete, the current node and current node list revert to what they were before the instruction was instantiated.

XSLT makes use of the expression language defined by [XPath] for selecting elements for processing, for conditional processing and for generating text.

XSLT provides two "hooks" for extending the language, one hook for extending the set of instruction elements used in templates and one hook for extending the set of functions used in XPath expressions. These hooks are both based on XML namespaces. This version of XSLT does not define a mechanism for implementing the hooks. See [**14 Extensions**].

> **NOTE:** The XSL WG intends to define such a mechanism in a future version of this specification or in a separate specification.

The element syntax summary notation used to describe the syntax of XSLT-defined elements is described in [**18 Notation**].

The MIME media types `text/xml` and `application/xml` [RFC2376] should be used for XSLT stylesheets. It is possible that a media type will be registered specifically for XSLT stylesheets; if and when it is, that media type may also be used.

2 Stylesheet Structure

2.1 XSLT Namespace

The XSLT namespace has the URI `http://www.w3.org/1999/XSL/Transform`.

> **NOTE:** The `1999` in the URI indicates the year in which the URI was allocated by the W3C. It does not indicate the version of XSLT being used, which is specified by attributes (see [**2.2 Stylesheet Element**] and [**2.3 Literal Result Element as Stylesheet**]).

XSLT processors must use the XML namespaces mechanism [XML Names] to recognize elements and attributes from this namespace. Elements from the XSLT namespace are recognized only in the stylesheet not in the source document. The complete list of XSLT-defined elements is specified in [**B Element Syntax Summary**]. Vendors must not extend the XSLT namespace with additional elements or attributes.

Instead, any extension must be in a separate namespace. Any namespace that is used for additional instruction elements must be identified by means of the extension element mechanism specified in [14.1 Extension Elements].

This specification uses a prefix of `xsl:` for referring to elements in the XSLT namespace. However, XSLT stylesheets are free to use any prefix, provided that there is a namespace declaration that binds the prefix to the URI of the XSLT namespace.

An element from the XSLT namespace may have any attribute not from the XSLT namespace, provided that the expanded-name of the attribute has a non-null namespace URI. The presence of such attributes must not change the behavior of XSLT elements and functions defined in this document. Thus, an XSLT processor is always free to ignore such attributes, and must ignore such attributes without giving an error if it does not recognize the namespace URI. Such attributes can provide, for example, unique identifiers, optimization hints, or documentation.

It is an error for an element from the XSLT namespace to have attributes with expanded-names that have null namespace URIs (i.e. attributes with unprefixed names) other than attributes defined for the element in this document.

> **NOTE:** The conventions used for the names of XSLT elements, attributes and functions are that names are all lower-case, use hyphens to separate words, and use abbreviations only if they already appear in the syntax of a related language such as XML or HTML.

2.2 Stylesheet Element

```
<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:stylesheet>
```

```
<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:transform>
```

A stylesheet is represented by an `xsl:stylesheet` element in an XML document. `xsl:transform` is allowed as a synonym for `xsl:stylesheet`.

An `xsl:stylesheet` element must have a `version` attribute, indicating the version of XSLT that the stylesheet requires. For this version of XSLT, the value should be `1.0`. When the value is not equal to `1.0`, forwards-compatible processing mode is enabled (see [2.5 Forwards-Compatible Processing]).

The `xsl:stylesheet` element may contain the following types of elements:

- `xsl:import`
- `xsl:include`
- `xsl:strip-space`
- `xsl:preserve-space`
- `xsl:output`
- `xsl:key`
- `xsl:decimal-format`
- `xsl:namespace-alias`
- `xsl:attribute-set`
- `xsl:variable`
- `xsl:param`
- `xsl:template`

An element occurring as a child of an `xsl:stylesheet` element is called a **top-level** element.

This example shows the structure of a stylesheet. Ellipses (`...`) indicate where attribute values or content have been omitted. Although this example shows one of each type of allowed element, stylesheets may contain zero or more of each of these elements.

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="..."/>

  <xsl:include href="..."/>

  <xsl:strip-space elements="..."/>

  <xsl:preserve-space elements="..."/>

  <xsl:output method="..."/>

  <xsl:key name="..." match="..." use="..."/>

  <xsl:decimal-format name="..."/>

  <xsl:namespace-alias stylesheet-prefix="..." result-prefix="..."/>

  <xsl:attribute-set name="...">
    ...
  </xsl:attribute-set>

  <xsl:variable name="...">...</xsl:variable>

  <xsl:param name="...">...</xsl:param>

  <xsl:template match="...">
    ...
  </xsl:template>

  <xsl:template name="...">
    ...
  </xsl:template>

</xsl:stylesheet>
```

The order in which the children of the `xsl:stylesheet` element occur is not significant except for `xsl:import` elements and for error recovery. Users are free to order the elements as they prefer, and stylesheet creation tools need not provide control over the order in which the elements occur.

In addition, the `xsl:stylesheet` element may contain any element not from the XSLT namespace, provided that the expanded-name of the element has a non-null namespace URI. The presence of such top-level elements must not change the behavior of XSLT elements and functions defined in this document; for example, it would not be permitted for such a top-level element to specify that `xsl:apply-templates` was to use different rules to resolve conflicts. Thus, an XSLT processor is always free to ignore such top-level elements, and must ignore a top-level element without giving an error if it does not recognize the namespace URI. Such elements can provide, for example,

- information used by extension elements or extension functions (see [14 Extensions]),
- information about what to do with the result tree,
- information about how to obtain the source tree,
- metadata about the stylesheet,
- structured documentation for the stylesheet.

2.3 Literal Result Element as Stylesheet

A simplified syntax is allowed for stylesheets that consist of only a single template for the root node. The stylesheet may consist of just a literal result element (see [7.1.1 Literal Result Elements]). Such a stylesheet is equivalent to a stylesheet with an `xsl:stylesheet` element containing a template rule containing the literal result element; the template rule has a match pattern of `/`. For example

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.w3.org/TR/xhtml1/strict">
  <head>
    <title>Expense Report Summary</title>
```

```
  </head>
  <body>
    <p>Total Amount: <xsl:value-of select="expense-report/total"/></p>
  </body>
</html>
```

has the same meaning as

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match="/">
<html>
  <head>
    <title>Expense Report Summary</title>
  </head>
  <body>
    <p>Total Amount: <xsl:value-of select="expense-report/total"/></p>
  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

A literal result element that is the document element of a stylesheet must have an `xsl:version` attribute, which indicates the version of XSLT that the stylesheet requires. For this version of XSLT, the value should be `1.0`; the value must be a Number. Other literal result elements may also have an `xsl:version` attribute. When the `xsl:version` attribute is not equal to `1.0`, forwards-compatible processing mode is enabled (see [2.5 Forwards-Compatible Processing]).

The allowed content of a literal result element when used as a stylesheet is no different from when it occurs within a stylesheet. Thus, a literal result element used as a stylesheet cannot contain top-level elements.

In some situations, the only way that a system can recognize that an XML document needs to be processed by an XSLT processor as an XSLT stylesheet is by examining the XML document itself. Using the simplified syntax makes this harder.

> **NOTE:**For example, another XML language (AXL) might also use an `axl:version` on the document element to indicate that an XML document was an AXL document that required processing by an AXL processor; if a document had both an `axl:version` attribute and an `xsl:version` attribute, it would be unclear whether the document should be processed by an XSLT processor or an AXL processor.

Therefore, the simplified syntax should not be used for XSLT stylesheets that may be used in such a situation. This situation can, for example, arise when an XSLT stylesheet is transmitted as a message with a MIME media type of `text/xml` or `application/xml` to a recipient that will use the MIME media type to determine how the message is processed.

2.4 Qualified Names

The name of an internal XSLT object, specifically a named template (see [6 Named Templates]), a mode (see [5.7 Modes]), an attribute set (see [7.1.4 Named Attribute Sets]), a key (see [12.2 Keys]), a decimal-format (see [12.3 Number Formatting]), a variable or a parameter (see [11 Variables and Parameters]) is specified as a QName. If it has a prefix, then the prefix is expanded into a URI reference using the namespace declarations in effect on the attribute in which the name occurs. The expanded-name consisting of the local part of the name and the possibly null URI reference is used as the name of the object. The default namespace is *not* used for unprefixed names.

2.5 Forwards-Compatible Processing

An element enables forwards-compatible mode for itself, its attributes, its descendants and their attributes if either it is an `xsl:stylesheet` element whose `version` attribute is not equal to `1.0`, or it is a literal result element that has an `xsl:version` attribute whose value is not equal to `1.0`, or it is a literal result element that does not have an `xsl:version` attribute and that is the document element of a stylesheet using the simplified syntax (see [2.3 Literal Result Element as Stylesheet]). A literal result element that has an `xsl:version` attribute whose value is equal to `1.0` disables forwards-compatible mode for itself, its attributes, its descendants and their attributes.

If an element is processed in forwards-compatible mode, then:

- if it is a top-level element and XSLT 1.0 does not allow such elements as top-level elements, then the element must be ignored along with its content;
- if it is an element in a template and XSLT 1.0 does not allow such elements to occur in templates, then if the element is not instantiated, an error must not be signaled, and if the element is instantiated, the XSLT must perform fallback for the element as specified in [15 Fallback];
- if the element has an attribute that XSLT 1.0 does not allow the element to have or if the element has an optional attribute with a value that the XSLT 1.0 does not allow the attribute to have, then the attribute must be ignored.

Thus, any XSLT 1.0 processor must be able to process the following stylesheet without error, although the stylesheet includes elements from the XSLT namespace that are not defined in this specification:

```
<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:choose>
      <xsl:when test="system-property('xsl:version') >= 1.1">
        <xsl:exciting-new-1.1-feature/>
      </xsl:when>
      <xsl:otherwise>
        <html>
        <head>
          <title>XSLT 1.1 required</title>
        </head>
        <body>
          <p>Sorry, this stylesheet requires XSLT 1.1.</p>
        </body>
        </html>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
</xsl:stylesheet>
```

> **NOTE:**If a stylesheet depends crucially on a top-level element introduced by a version of XSL after 1.0, then the stylesheet can use an `xsl:message` element with `terminate="yes"` (see [13 Messages]) to ensure that XSLT processors implementing earlier versions of XSL will not silently ignore the top-level element. For example,
>
> ```
> <xsl:stylesheet version="1.5"
>                 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
>
>   <xsl:important-new-1.1-declaration/>
>
>   <xsl:template match="/">
>     <xsl:choose>
>       <xsl:when test="system-property('xsl:version') < 1.1">
>         <xsl:message terminate="yes">
>           <xsl:text>Sorry, this stylesheet requires XSLT 1.1.</xsl:text>
>         </xsl:message>
>       </xsl:when>
>       <xsl:otherwise>
>         ...
>       </xsl:otherwise>
>     </xsl:choose>
>   </xsl:template>
>   ...
> </xsl:stylesheet>
> ```

If an expression occurs in an attribute that is processed in forwards-compatible mode, then an XSLT processor must recover from errors in the expression as follows:

- if the expression does not match the syntax allowed by the XPath grammar, then an error must not be signaled unless the expression is actually evaluated;
- if the expression calls a function with an unprefixed name that is not part of the XSLT library, then an error must not be signaled unless the function is actually called;
- if the expression calls a function with a number of arguments that XSLT does not allow or with arguments of types that XSLT does not allow, then an error must not be signaled unless the function is actually called.

2.6 Combining Stylesheets

XSLT provides two mechanisms to combine stylesheets:

- an inclusion mechanism that allows stylesheets to be combined without changing the semantics of the stylesheets being combined, and
- an import mechanism that allows stylesheets to override each other.

2.6.1 Stylesheet Inclusion

```
<!-- Category: top-level-element -->
<xsl:include
  href = uri-reference />
```

An XSLT stylesheet may include another XSLT stylesheet using an `xsl:include` element. The `xsl:include` element has an `href` attribute whose value is a URI reference identifying the stylesheet to be included. A relative URI is resolved relative to the base URI of the `xsl:include` element (see [3.2 Base URI]).

The `xsl:include` element is only allowed as a top-level element.

The inclusion works at the XML tree level. The resource located by the `href` attribute value is parsed as an XML document, and the children of the `xsl:stylesheet` element in this document replace the `xsl:include` element in the including document. The fact that template rules or definitions are included does not affect the way they are processed.

The included stylesheet may use the simplified syntax described in [2.3 Literal Result Element as Stylesheet]. The included stylesheet is treated the same as the equivalent `xsl:stylesheet` element.

It is an error if a stylesheet directly or indirectly includes itself.

> **NOTE:**Including a stylesheet multiple times can cause errors because of duplicate definitions. Such multiple inclusions are less obvious when they are indirect. For example, if stylesheet *B* includes stylesheet *A*, stylesheet *C* includes stylesheet *A*, and stylesheet *D* includes both stylesheet *B* and stylesheet *C*, then *A* will be included indirectly by *D* twice. If all of *B*, *C* and *D* are used as independent stylesheets, then the error can be avoided by separating everything in *B* other than the inclusion of *A* into a separate stylesheet *B'* and changing *B* to contain just inclusions of *B'* and *A*, similarly for *C*, and then changing *D* to include *A*, *B'*, *C'*.

2.6.2 Stylesheet Import

```
<xsl:import
  href = uri-reference />
```

An XSLT stylesheet may import another XSLT stylesheet using an `xsl:import` element. Importing a stylesheet is the same as including it (see [2.6.1 Stylesheet Inclusion]) except that definitions and template rules in the importing stylesheet take precedence over template rules and definitions in the imported stylesheet; this is described in more detail below. The `xsl:import` element has an `href` attribute whose value is a URI reference identifying the stylesheet to be imported. A relative URI is resolved relative to the base URI of the `xsl:import` element (see [3.2 Base URI]).

The `xsl:import` element is only allowed as a top-level element. The `xsl:import` element children must precede all other element children of an `xsl:stylesheet` element, including any `xsl:include` element children. When `xsl:include` is used to include a stylesheet, any `xsl:import` elements in the included document are moved up in the including document to after any existing `xsl:import` elements in the including document.

For example,

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:import href="article.xsl"/>
  <xsl:import href="bigfont.xsl"/>
  <xsl:attribute-set name="note-style">
    <xsl:attribute name="font-style">italic</xsl:attribute>
  </xsl:attribute-set>
</xsl:stylesheet>
```

The `xsl:stylesheet` elements encountered during processing of a stylesheet that contains `xsl:import` elements are treated as forming an **import tree**. In the import tree, each `xsl:stylesheet` element has one import child for each `xsl:import` element that it contains. Any `xsl:include` elements are resolved before

constructing the import tree. An `xsl:stylesheet` element in the import tree is defined to have lower **import precedence** than another `xsl:stylesheet` element in the import tree if it would be visited before that `xsl:stylesheet` element in a post-order traversal of the import tree (i.e. a traversal of the import tree in which an `xsl:stylesheet` element is visited after its import children). Each definition and template rule has import precedence determined by the `xsl:stylesheet` element that contains it.

For example, suppose

- stylesheet *A* imports stylesheets *B* and *C* in that order;
- stylesheet *B* imports stylesheet *D*;
- stylesheet *C* imports stylesheet *E*.

Then the order of import precedence (lowest first) is *D*, *B*, *E*, *C*, *A*.

> **NOTE:**Since `xsl:import` elements are required to occur before any definitions or template rules, an implementation that processes imported stylesheets at the point at which it encounters the `xsl:import` element will encounter definitions and template rules in increasing order of import precedence.

In general, a definition or template rule with higher import precedence takes precedence over a definition or template rule with lower import precedence. This is defined in detail for each kind of definition and for template rules.

It is an error if a stylesheet directly or indirectly imports itself. Apart from this, the case where a stylesheet with a particular URI is imported in multiple places is not treated specially. The import tree will have a separate `xsl:stylesheet` for each place that it is imported.

> **NOTE:**If `xsl:apply-imports` is used (see [5.6 Overriding Template Rules]), the behavior may be different from the behavior if the stylesheet had been imported only at the place with the highest import precedence.

2.7 Embedding Stylesheets

Normally an XSLT stylesheet is a complete XML document with the `xsl:stylesheet` element as the document element. However, an XSLT stylesheet may also be embedded in another resource. Two forms of embedding are possible:

- the XSLT stylesheet may be textually embedded in a non-XML resource, or
- the `xsl:stylesheet` element may occur in an XML document other than as the document element.

To facilitate the second form of embedding, the `xsl:stylesheet` element is allowed to have an ID attribute that specifies a unique identifier.

> **NOTE:**In order for such an attribute to be used with the XPath id function, it must actually be declared in the DTD as being an ID.

The following example shows how the `xml-stylesheet` processing instruction [XML Stylesheet] can be used to allow a document to contain its own stylesheet. The URI reference uses a relative URI with a fragment identifier to locate the `xsl:stylesheet` element:

```
<?xml-stylesheet type="text/xml" href="#style1"?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<head>
<xsl:stylesheet id="style1"
                version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format">
<xsl:import href="doc.xsl"/>
<xsl:template match="id('foo')">
  <fo:block font-weight="bold"><xsl:apply-templates/></fo:block>
</xsl:template>
<xsl:template match="xsl:stylesheet">
  <!-- ignore -->
</xsl:template>
</xsl:stylesheet>
</head>
```

```
<body>
<para id="foo">
...
</para>
</body>
</doc>
```

NOTE: A stylesheet that is embedded in the document to which it is to be applied or that may be included or imported into an stylesheet that is so embedded typically needs to contain a template rule that specifies that `xsl:stylesheet` elements are to be ignored.

3 Data Model

The data model used by XSLT is the same as that used by XPath with the additions described in this section. XSLT operates on source, result and stylesheet documents using the same data model. Any two XML documents that have the same tree will be treated the same by XSLT.

Processing instructions and comments in the stylesheet are ignored: the stylesheet is treated as if neither processing instruction nodes nor comment nodes were included in the tree that represents the stylesheet.

3.1 Root Node Children

The normal restrictions on the children of the root node are relaxed for the result tree. The result tree may have any sequence of nodes as children that would be possible for an element node. In particular, it may have text node children, and any number of element node children. When written out using the XML output method (see [16 Output]), it is possible that a result tree will not be a well-formed XML document; however, it will always be a well-formed external general parsed entity.

When the source tree is created by parsing a well-formed XML document, the root node of the source tree will automatically satisfy the normal restrictions of having no text node children and exactly one element child. When the source tree is created in some other way, for example by using the DOM, the usual restrictions are relaxed for the source tree as for the result tree.

3.2 Base URI

Every node also has an associated URI called its base URI, which is used for resolving attribute values that represent relative URIs into absolute URIs. If an element or processing instruction occurs in an external entity, the base URI of that element or processing instruction is the URI of the external entity; otherwise, the base URI is the base URI of the document. The base URI of the document node is the URI of the document entity. The base URI for a text node, a comment node, an attribute node or a namespace node is the base URI of the parent of the node.

3.3 Unparsed Entities

The root node has a mapping that gives the URI for each unparsed entity declared in the document's DTD. The URI is generated from the system identifier and public identifier specified in the entity declaration. The XSLT processor may use the public identifier to generate a URI for the entity instead of the URI specified in the system identifier. If the XSLT processor does not use the public identifier to generate the URI, it must use the system identifier; if the system identifier is a relative URI, it must be resolved into an absolute URI using the URI of the resource containing the entity declaration as the base URI [RFC2396].

3.4 Whitespace Stripping

After the tree for a source document or stylesheet document has been constructed, but before it is otherwise processed by XSLT, some text nodes are stripped. A text node is never stripped unless it contains only whitespace characters. Stripping the text node removes the text node from the tree. The stripping process takes as input a set of element names for which whitespace must be preserved. The stripping process is applied to both stylesheets and source documents, but the set of whitespace-preserving element names is determined differently for stylesheets and for source documents.

A text node is preserved if any of the following apply:

- The element name of the parent of the text node is in the set of whitespace-preserving element names.

- The text node contains at least one non-whitespace character. As in XML, a whitespace character is #x20, #x9, #xD or #xA.
- An ancestor element of the text node has an `xml:space` attribute with a value of `preserve`, and no closer ancestor element has `xml:space` with a value of `default`.

Otherwise, the text node is stripped.

The `xml:space` attributes are not stripped from the tree.

NOTE: This implies that if an `xml:space` attribute is specified on a literal result element, it will be included in the result.

For stylesheets, the set of whitespace-preserving element names consists of just `xsl:text`.

```
<!-- Category: top-level-element -->
<xsl:strip-space
  elements = tokens />
```

```
<!-- Category: top-level-element -->
<xsl:preserve-space
  elements = tokens />
```

For source documents, the set of whitespace-preserving element names is specified by `xsl:strip-space` and `xsl:preserve-space` top-level elements. These elements each have an `elements` attribute whose value is a whitespace-separated list of NameTests. Initially, the set of whitespace-preserving element names contains all element names. If an element name matches a NameTest in an `xsl:strip-space` element, then it is removed from the set of whitespace-preserving element names. If an element name matches a NameTest in an `xsl:preserve-space` element, then it is added to the set of whitespace-preserving element names. An element matches a NameTest if and only if the NameTest would be true for the element as an XPath node test. Conflicts between matches to `xsl:strip-space` and `xsl:preserve-space` elements are resolved the same way as conflicts between template rules (see [5.5 Conflict Resolution for Template Rules]). Thus, the applicable match for a particular element name is determined as follows:

- First, any match with lower import precedence than another match is ignored.
- Next, any match with a NameTest that has a lower default priority than the default priority of the NameTest of another match is ignored.

It is an error if this leaves more than one match. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the matches that are left, the one that occurs last in the stylesheet.

4 Expressions

XSLT uses the expression language defined by XPath [XPath]. Expressions are used in XSLT for a variety of purposes including:

- selecting nodes for processing;
- specifying conditions for different ways of processing a node;
- generating text to be inserted in the result tree.

An **expression** must match the XPath production Expr.

Expressions occur as the value of certain attributes on XSLT-defined elements and within curly braces in attribute value templates.

In XSLT, an outermost expression (i.e. an expression that is not part of another expression) gets its context as follows:

- the context node comes from the current node
- the context position comes from the position of the current node in the current node list; the first position is 1

- the context size comes from the size of the current node list
- the variable bindings are the bindings in scope on the element which has the attribute in which the expression occurs (see [11 Variables and Parameters])
- the set of namespace declarations are those in scope on the element which has the attribute in which the expression occurs; this includes the implicit declaration of the prefix `xml` required by the the XML Namespaces Recommendation [XML Names]; the default namespace (as declared by `xmlns`) is not part of this set
- the function library consists of the core function library together with the additional functions defined in [12 Additional Functions] and extension functions as described in [14 Extensions]; it is an error for an expression to include a call to any other function

5 Template Rules

5.1 Processing Model

A list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.

Implementations are free to process the source document in any way that produces the same result as if it were processed using this processing model.

5.2 Patterns

Template rules identify the nodes to which they apply by using a **pattern**. As well as being used in template rules, patterns are used for numbering (see [7.7 Numbering]) and for declaring keys (see [12.2 Keys]). A pattern specifies a set of conditions on a node. A node that satisfies the conditions matches the pattern; a node that does not satisfy the conditions does not match the pattern. The syntax for patterns is a subset of the syntax for expressions. In particular, location paths that meet certain restrictions can be used as patterns. An expression that is also a pattern always evaluates to an object of type node-set. A node matches a pattern if the node is a member of the result of evaluating the pattern as an expression with respect to some possible context; the possible contexts are those whose context node is the node being matched or one of its ancestors.

Here are some examples of patterns:

- `para` matches any `para` element
- `*` matches any element
- `chapter|appendix` matches any `chapter` element and any `appendix` element
- `olist/item` matches any `item` element with an `olist` parent
- `appendix//para` matches any `para` element with an `appendix` ancestor element
- `/` matches the root node
- `text()` matches any text node
- `processing-instruction()` matches any processing instruction
- `node()` matches any node other than an attribute node and the root node
- `id("W11")` matches the element with unique ID `W11`

- `para[1]` matches any `para` element that is the first `para` child element of its parent
- `*[position()=1 and self::para]` matches any `para` element that is the first child element of its parent
- `para[last()=1]` matches any `para` element that is the only `para` child element of its parent
- `items/item[position()>1]` matches any `item` element that has a `items` parent and that is not the first `item` child of its parent
- `item[position() mod 2 = 1]` would be true for any `item` element that is an odd-numbered `item` child of its parent.
- `div[@class="appendix"]//p` matches any `p` element with a `div` ancestor element that has a `class` attribute with value `appendix`
- `@class` matches any `class` attribute (*not* any element that has a `class` attribute)
- `@*` matches any attribute

A pattern must match the grammar for Pattern. A Pattern is a set of location path patterns separated by `|`. A location path pattern is a location path whose steps all use only the `child` or `attribute` axes. Although patterns must not use the `descendant-or-self` axis, patterns may use the `//` operator as well as the `/` operator. Location path patterns can also start with an **id** or **key** function call with a literal argument. Predicates in a pattern can use arbitrary expressions just like predicates in a location path.

Patterns

[1]	Pattern	::=	LocationPathPattern
			| Pattern '|' LocationPathPattern
[2]	LocationPathPattern	::=	'/' RelativePathPattern?
			| IdKeyPattern (('/' | '//') RelativePathPattern)?
			| '//'? RelativePathPattern
[3]	IdKeyPattern	::=	'id' '(' Literal ')'
			| 'key' '(' Literal ',' Literal ')'
[4]	RelativePathPattern	::=	StepPattern
			| RelativePathPattern '/' StepPattern
			| RelativePathPattern '//' StepPattern
[5]	StepPattern	::=	ChildOrAttributeAxisSpecifier NodeTest Predicate*
[6]	ChildOrAttributeAxisSpecifier	::=	AbbreviatedAxisSpecifier
			| ('child' | 'attribute') '::'

A pattern is defined to match a node if and only if there is possible context such that when the pattern is evaluated as an expression with that context, the node is a member of the resulting node-set. When a node is being matched, the possible contexts have a context node that is the node being matched or any ancestor of that node, and a context node list containing just the context node.

For example, `p` matches any `p` element, because for any `p` if the expression `p` is evaluated with the parent of the `p` element as context the resulting node-set will contain that `p` element as one of its members.

> **NOTE:**This matches even a `p` element that is the document element, since the document root is the parent of the document element.

Although the semantics of patterns are specified indirectly in terms of expression evaluation, it is easy to understand the meaning of a pattern directly without thinking in terms of expression evaluation. In a pattern, `|` indicates alternatives; a pattern with one or more `|` separated alternatives matches if any one of the alternative matches. A pattern that consists of a sequence of StepPatterns separated by `/` or `//` is matched from right to left. The pattern only matches if the rightmost StepPattern matches and a suitable element matches the rest of the pattern; if the separator is `/` then only the parent is a suitable element; if the

separator is `//`, then any ancestor is a suitable element. A StepPattern that uses the child axis matches if the NodeTest is true for the node and the node is not an attribute node. A StepPattern that uses the attribute axis matches if the NodeTest is true for the node and the node is an attribute node. When `[]` is present, then the first PredicateExpr in a StepPattern is evaluated with the node being matched as the context node and the siblings of the context node that match the NodeTest as the context node list, unless the node being matched is an attribute node, in which case the context node list is all the attributes that have the same parent as the attribute being matched and that match the NameTest.

For example

```
appendix//ulist/item[position()=1]
```

matches a node if and only if all of the following are true:

- the NodeTest `item` is true for the node and the node is not an attribute; in other words the node is an `item` element
- evaluating the PredicateExpr `position()=1` with the node as context node and the siblings of the node that are `item` elements as the context node list yields true
- the node has a parent that matches `appendix//ulist`; this will be true if the parent is a `ulist` element that has an `appendix` ancestor element.

5.3 Defining Template Rules

```
<!-- Category: top-level-element -->
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- Content: (xsl:param*, template) -->
</xsl:template>
```

A template rule is specified with the `xsl:template` element. The `match` attribute is a Pattern that identifies the source node or nodes to which the rule applies. The `match` attribute is required unless the `xsl:template` element has a `name` attribute (see [6 Named Templates]). It is an error for the value of the `match` attribute to contain a VariableReference. The content of the `xsl:template` element is the template that is instantiated when the template rule is applied.

For example, an XML document might contain:

```
This is an <emph>important</emph> point.
```

The following template rule matches `emph` elements and produces a `fo:inline-sequence` formatting object with a `font-weight` property of `bold`.

```
<xsl:template match="emph">
  <fo:inline-sequence font-weight="bold">
    <xsl:apply-templates/>
  </fo:inline-sequence>
</xsl:template>
```

NOTE: Examples in this document use the `fo:` prefix for the namespace `http://www.w3.org/1999/XSL/Format`, which is the namespace of the formatting objects defined in [XSL].

As described next, the `xsl:apply-templates` element recursively processes the children of the source element.

5.4 Applying Template Rules

```
<!-- Category: instruction -->
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

This example creates a block for a `chapter` element and then processes its immediate children.

```
<xsl:template match="chapter">
  <fo:block>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

In the absence of a `select` attribute, the `xsl:apply-templates` instruction processes all of the children of the current node, including text nodes. However, text nodes that have been stripped as specified in [3.4 Whitespace Stripping] will not be processed. If stripping of whitespace nodes has not been enabled for an element, then all whitespace in the content of the element will be processed as text, and thus whitespace between child elements will count in determining the position of a child element as returned by the **position** function.

A `select` attribute can be used to process nodes selected by an expression instead of processing all children. The value of the `select` attribute is an expression. The expression must evaluate to a node-set. The selected set of nodes is processed in document order, unless a sorting specification is present (see [10 Sorting]). The following example processes all of the `author` children of the `author-group`:

```
<xsl:template match="author-group">
  <fo:inline-sequence>
    <xsl:apply-templates select="author"/>
  </fo:inline-sequence>
</xsl:template>
```

The following example processes all of the `given-name`s of the `author`s that are children of `author-group`:

```
<xsl:template match="author-group">
  <fo:inline-sequence>
    <xsl:apply-templates select="author/given-name"/>
  </fo:inline-sequence>
</xsl:template>
```

This example processes all of the `heading` descendant elements of the `book` element.

```
<xsl:template match="book">
  <fo:block>
    <xsl:apply-templates select=".//heading"/>
  </fo:block>
</xsl:template>
```

It is also possible to process elements that are not descendants of the current node. This example assumes that a `department` element has `group` children and `employee` descendants. It finds an employee's department and then processes the `group` children of the `department`.

```
<xsl:template match="employee">
  <fo:block>
    Employee <xsl:apply-templates select="name"/> belongs to group
    <xsl:apply-templates select="ancestor::department/group"/>
  </fo:block>
</xsl:template>
```

Multiple `xsl:apply-templates` elements can be used within a single template to do simple reordering. The following example creates two HTML tables. The first table is filled with domestic sales while the second table is filled with foreign sales.

```
<xsl:template match="product">
  <table>
    <xsl:apply-templates select="sales/domestic"/>
  </table>
  <table>
    <xsl:apply-templates select="sales/foreign"/>
  </table>
</xsl:template>
```

NOTE: It is possible for there to be two matching descendants where one is a descendant of the other. This case is not treated specially: both descendants will be processed as usual. For example, given a source document

```
<doc><div><div></div></div></doc>
```

the rule

```
<xsl:template match="doc">
  <xsl:apply-templates select=".//div"/>
</xsl:template>
```

will process both the outer `div` and inner `div` elements.

NOTE: Typically, `xsl:apply-templates` is used to process only nodes that are descendants of the current node. Such use of `xsl:apply-templates` cannot result in non-terminating processing loops. However, when `xsl:apply-templates` is used to process elements that are not descendants of the current node, the possibility arises of non-terminating loops. For example,

```
<xsl:template match="foo">
  <xsl:apply-templates select="."/>
</xsl:template>
```

Implementations may be able to detect such loops in some cases, but the possibility exists that a stylesheet may enter a non-terminating loop that an implementation is unable to detect. This may present a denial of service security risk.

5.5 Conflict Resolution for Template Rules

It is possible for a source node to match more than one template rule. The template rule to be used is determined as follows:

1. First, all matching template rules that have lower import precedence than the matching template rule or rules with the highest import precedence are eliminated from consideration.
2. Next, all matching template rules that have lower priority than the matching template rule or rules with the highest priority are eliminated from consideration. The priority of a template rule is specified by the `priority` attribute on the template rule. The value of this must be a real number (positive or negative), matching the production Number with an optional leading minus sign (-). The **default priority** is computed as follows:
   - If the pattern contains multiple alternatives separated by `|`, then it is treated equivalently to a set of template rules, one for each alternative.
   - If the pattern has the form of a QName preceded by a ChildOrAttributeAxisSpecifier or has the form `processing-instruction(`Literal`)` preceded by a ChildOrAttributeAxisSpecifier, then the priority is 0.
   - If the pattern has the form NCName`:*` preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.25.
   - Otherwise, if the pattern consists of just a NodeTest preceded by a ChildOrAttributeAxisSpecifier, then the priority is -0.5.
   - Otherwise, the priority is 0.5.

   Thus, the most common kind of pattern (a pattern that tests for a node with a particular type and a particular expanded-name) has priority 0. The next less specific kind of pattern (a pattern that tests for a node with a particular type and an expanded-name with a particular namespace URI) has priority -0.25. Patterns less specific than this (patterns that just tests for nodes with particular types) have priority -0.5. Patterns more specific than the most common kind of pattern have priority 0.5.

It is an error if this leaves more than one matching template rule. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the matching template rules that are left, the one that occurs last in the stylesheet.

5.6 Overriding Template Rules

```
<!-- Category: instruction -->
<xsl:apply-imports />
```

A template rule that is being used to override a template rule in an imported stylesheet (see **[5.5 Conflict Resolution for Template Rules]**) can use the `xsl:apply-imports` element to invoke the overridden template rule.

At any point in the processing of a stylesheet, there is a **current template rule**. Whenever a template rule is chosen by matching a pattern, the template rule becomes the current template rule for the instantiation of the rule's template. When an `xsl:for-each` element is instantiated, the current template rule becomes null for

the instantiation of the content of the `xsl:for-each` element.

`xsl:apply-imports` processes the current node using only template rules that were imported into the stylesheet element containing the current template rule; the node is processed in the current template rule's mode. It is an error if `xsl:apply-imports` is instantiated when the current template rule is null.

For example, suppose the stylesheet `doc.xsl` contains a template rule for `example` elements:

```
<xsl:template match="example">
  <pre><xsl:apply-templates/></pre>
</xsl:template>
```

Another stylesheet could import `doc.xsl` and modify the treatment of `example` elements as follows:

```
<xsl:import href="doc.xsl"/>

<xsl:template match="example">
  <div style="border: solid red">
    <xsl:apply-imports/>
  </div>
</xsl:template>
```

The combined effect would be to transform an `example` into an element of the form:

```
<div style="border: solid red"><pre>...</pre></div>
```

5.7 Modes

Modes allow an element to be processed multiple times, each time producing a different result.

Both `xsl:template` and `xsl:apply-templates` have an optional `mode` attribute. The value of the `mode` attribute is a QName, which is expanded as described in **[2.4 Qualified Names]**. If `xsl:template` does not have a `match` attribute, it must not have a `mode` attribute. If an `xsl:apply-templates` element has a `mode` attribute, then it applies only to those template rules from `xsl:template` elements that have a `mode` attribute with the same value; if an `xsl:apply-templates` element does not have a `mode` attribute, then it applies only to those template rules from `xsl:template` elements that do not have a `mode` attribute.

5.8 Built-in Template Rules

There is a built-in template rule to allow recursive processing to continue in the absence of a successful pattern match by an explicit template rule in the stylesheet. This template rule applies to both element nodes and the root node. The following shows the equivalent of the built-in template rule:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

There is also a built-in template rule for each mode, which allows recursive processing to continue in the same mode in the absence of a successful pattern match by an explicit template rule in the stylesheet. This template rule applies to both element nodes and the root node. The following shows the equivalent of the built-in template rule for mode *m*.

```
<xsl:template match="*|/" mode="m">
  <xsl:apply-templates mode="m"/>
</xsl:template>
```

There is also a built-in template rule for text and attribute nodes that copies text through:

```
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

The built-in template rule for processing instructions and comments is to do nothing.

```
<xsl:template match="processing-instruction()|comment()"/>
```

The built-in template rule for namespace nodes is also to do nothing. There is no pattern that can match a namespace node; so, the built-in template rule is the only template rule that is applied for namespace nodes.

The built-in template rules are treated as if they were imported implicitly before the stylesheet and so have lower import precedence than all other template rules. Thus, the author can override a built-in template rule by

including an explicit template rule.

6 Named Templates

```
<!-- Category: instruction -->
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>
```

Templates can be invoked by name. An `xsl:template` element with a `name` attribute specifies a named template. The value of the `name` attribute is a QName, which is expanded as described in [2.4 Qualified Names]. If an `xsl:template` element has a `name` attribute, it may, but need not, also have a `match` attribute. An `xsl:call-template` element invokes a template by name; it has a required `name` attribute that identifies the template to be invoked. Unlike `xsl:apply-templates`, `xsl:call-template` does not change the current node or the current node list.

The `match`, `mode` and `priority` attributes on an `xsl:template` element do not affect whether the template is invoked by an `xsl:call-template` element. Similarly, the `name` attribute on an `xsl:template` element does not affect whether the template is invoked by an `xsl:apply-templates` element.

It is an error if a stylesheet contains more than one template with the same name and same import precedence.

7 Creating the Result Tree

This section describes instructions that directly create nodes in the result tree.

7.1 Creating Elements and Attributes

7.1.1 Literal Result Elements

In a template, an element in the stylesheet that does not belong to the XSLT namespace and that is not an extension element (see [14.1 Extension Elements]) is instantiated to create an element node with the same expanded-name. The content of the element is a template, which is instantiated to give the content of the created element node. The created element node will have the attribute nodes that were present on the element node in the stylesheet tree, other than attributes with names in the XSLT namespace.

The created element node will also have a copy of the namespace nodes that were present on the element node in the stylesheet tree with the exception of any namespace node whose string-value is the XSLT namespace URI (`http://www.w3.org/1999/XSL/Transform`), a namespace URI declared as an extension namespace (see [14.1 Extension Elements]), or a namespace URI designated as an excluded namespace. A namespace URI is designated as an excluded namespace by using an `exclude-result-prefixes` attribute on an `xsl:stylesheet` element or an `xsl:exclude-result-prefixes` attribute on a literal result element. The value of both these attributes is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as an excluded namespace. It is an error if there is no namespace bound to the prefix on the element bearing the `exclude-result-prefixes` or `xsl:exclude-result-prefixes` attribute. The default namespace (as declared by `xmlns`) may be designated as an excluded namespace by including `#default` in the list of namespace prefixes. The designation of a namespace as an excluded namespace is effective within the subtree of the stylesheet rooted at the element bearing the `exclude-result-prefixes` or `xsl:exclude-result-prefixes` attribute; a subtree rooted at an `xsl:stylesheet` element does not include any stylesheets imported or included by children of that `xsl:stylesheet` element.

> **NOTE:**When a stylesheet uses a namespace declaration only for the purposes of addressing the source tree, specifying the prefix in the `exclude-result-prefixes` attribute will avoid superfluous namespace declarations in the result tree.

The value of an attribute of a literal result element is interpreted as an attribute value template: it can contain expressions contained in curly braces (`{}`).

A namespace URI in the stylesheet tree that is being used to specify a namespace URI in the result tree is called a **literal namespace URI**. This applies to:

- the namespace URI in the expanded-name of a literal result element in the stylesheet
- the namespace URI in the expanded-name of an attribute specified on a literal result element in the stylesheet
- the string-value of a namespace node on a literal result element in the stylesheet

```
<!-- Category: top-level-element -->
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

A stylesheet can use the `xsl:namespace-alias` element to declare that one namespace URI is an **alias** for another namespace URI. When a literal namespace URI has been declared to be an alias for another namespace URI, then the namespace URI in the result tree will be the namespace URI that the literal namespace URI is an alias for, instead of the literal namespace URI itself. The `xsl:namespace-alias` element declares that the namespace URI bound to the prefix specified by the `stylesheet-prefix` attribute is an alias for the namespace URI bound to the prefix specified by the `result-prefix` attribute. Thus, the `stylesheet-prefix` attribute specifies the namespace URI that will appear in the stylesheet, and the `result-prefix` attribute specifies the corresponding namespace URI that will appear in the result tree. The default namespace (as declared by `xmlns`) may be specified by using `#default` instead of a prefix. If a namespace URI is declared to be an alias for multiple different namespace URIs, then the declaration with the highest import precedence is used. It is an error if there is more than one such declaration. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing, from amongst the declarations with the highest import precedence, the one that occurs last in the stylesheet.

When literal result elements are being used to create element, attribute, or namespace nodes that use the XSLT namespace URI, the stylesheet must use an alias. For example, the stylesheet

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  xmlns:axsl="http://www.w3.org/1999/XSL/TransformAlias">

<xsl:namespace-alias stylesheet-prefix="axsl" result-prefix="xsl"/>

<xsl:template match="/">
  <axsl:stylesheet>
    <xsl:apply-templates/>
  </axsl:stylesheet>
</xsl:template>

<xsl:template match="block">
  <axsl:template match="{.}">
      <fo:block><axsl:apply-templates/></fo:block>
  </axsl:template>
</xsl:template>

</xsl:stylesheet>
```

will generate an XSLT stylesheet from a document of the form:

```
<elements>
<block>p</block>
<block>h1</block>
<block>h2</block>
<block>h3</block>
<block>h4</block>
</elements>
```

> **NOTE:**It may be necessary also to use aliases for namespaces other than the XSLT namespace URI. For example, literal result elements belonging to a namespace dealing with digital signatures might cause XSLT stylesheets to be mishandled by general-purpose security software; using an alias for the namespace would avoid the possibility of such mishandling.

7.1.2 Creating Elements with `xsl:element`

```
<!-- Category: instruction -->
<xsl:element
  name = { qname }
  namespace = { uri-reference }
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>
```

The `xsl:element` element allows an element to be created with a computed name. The expanded-name of the element to be created is specified by a required `name` attribute and an optional `namespace` attribute. The content of the `xsl:element` element is a template for the attributes and children of the created element.

The `name` attribute is interpreted as an attribute value template. It is an error if the string that results from instantiating the attribute value template is not a QName. An XSLT processor may signal the error; if it does not signal the error, then it must recover by making the the result of instantiating the `xsl:element` element be the sequence of nodes created by instantiating the content of the `xsl:element` element, excluding any initial attribute nodes. If the `namespace` attribute is not present then the QName is expanded into an expanded-name using the namespace declarations in effect for the `xsl:element` element, including any default namespace declaration.

If the `namespace` attribute is present, then it also is interpreted as an attribute value template. The string that results from instantiating the attribute value template should be a URI reference. It is not an error if the string is not a syntactically legal URI reference. If the string is empty, then the expanded-name of the element has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-name of the element to be created. The local part of the QName specified by the `name` attribute is used as the local part of the expanded-name of the element to be created.

XSLT processors may make use of the prefix of the QName specified in the `name` attribute when selecting the prefix used for outputting the created element as XML; however, they are not required to do so.

7.1.3 Creating Attributes with `xsl:attribute`

```
<!-- Category: instruction -->
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }>
  <!-- Content: template -->
</xsl:attribute>
```

The `xsl:attribute` element can be used to add attributes to result elements whether created by literal result elements in the stylesheet or by instructions such as `xsl:element`. The expanded-name of the attribute to be created is specified by a required `name` attribute and an optional `namespace` attribute. Instantiating an `xsl:attribute` element adds an attribute node to the containing result element node. The content of the `xsl:attribute` element is a template for the value of the created attribute.

The `name` attribute is interpreted as an attribute value template. It is an error if the string that results from instantiating the attribute value template is not a QName or is the string `xmlns`. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the attribute to the result tree. If the `namespace` attribute is not present, then the QName is expanded into an expanded-name using the namespace declarations in effect for the `xsl:attribute` element, *not* including any default namespace declaration.

If the `namespace` attribute is present, then it also is interpreted as an attribute value template. The string that results from instantiating it should be a URI reference. It is not an error if the string is not a syntactically legal URI reference. If the string is empty, then the expanded-name of the attribute has a null namespace URI. Otherwise, the string is used as the namespace URI of the expanded-name of the attribute to be created. The local part of the QName specified by the `name` attribute is used as the local part of the expanded-name of the attribute to be created.

XSLT processors may make use of the prefix of the QName specified in the `name` attribute when selecting the prefix used for outputting the created attribute as XML; however, they are not required to do so and, if the prefix is `xmlns`, they must not do so. Thus, although it is not an error to do:

```
<xsl:attribute name="xmlns:xsl" namespace="whatever">http://www.w3.org/1999/XSL/Transform</xsl:attribute>
```

it will not result in a namespace declaration being output.

Adding an attribute to an element replaces any existing attribute of that element with the same expanded-name.

The following are all errors:

- Adding an attribute to an element after children have been added to it; implementations may either signal the error or ignore the attribute.
- Adding an attribute to a node that is not an element; implementations may either signal the error or ignore the attribute.
- Creating nodes other than text nodes during the instantiation of the content of the `xsl:attribute` element; implementations may either signal the error or ignore the offending nodes.

  NOTE: When an `xsl:attribute` contains a text node with a newline, then the XML output must contain a character reference. For example,

  ```
  <xsl:attribute name="a">x
  y</xsl:attribute>
  ```

  will result in the output

  ```
  a="x&#xA;y"
  ```

  (or with any equivalent character reference). The XML output cannot be

  ```
  a="x
  y"
  ```

  This is because XML 1.0 requires newline characters in attribute values to be normalized into spaces but requires character references to newline characters not to be normalized. The attribute values in the data model represent the attribute value after normalization. If a newline occurring in an attribute value in the tree were output as a newline character rather than as character reference, then the attribute value in the tree created by reparsing the XML would contain a space not a newline, which would mean that the tree had not been output correctly.

7.1.4 Named Attribute Sets

```
<!-- Category: top-level-element -->
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

The `xsl:attribute-set` element defines a named set of attributes. The `name` attribute specifies the name of the attribute set. The value of the `name` attribute is a QName, which is expanded as described in [2.4 Qualified Names]. The content of the `xsl:attribute-set` element consists of zero or more `xsl:attribute` elements that specify the attributes in the set.

Attribute sets are used by specifying a `use-attribute-sets` attribute on `xsl:element`, `xsl:copy` (see [7.5 Copying]) or `xsl:attribute-set` elements. The value of the `use-attribute-sets` attribute is a whitespace-separated list of names of attribute sets. Each name is specified as a QName, which is expanded as described in [2.4 Qualified Names]. Specifying a `use-attribute-sets` attribute is equivalent to adding `xsl:attribute` elements for each of the attributes in each of the named attribute sets to the beginning of the content of the element with the `use-attribute-sets` attribute, in the same order in which the names of the attribute sets are specified in the `use-attribute-sets` attribute. It is an error if use of `use-attribute-sets` attributes on `xsl:attribute-set` elements causes an attribute set to directly or indirectly use itself.

Attribute sets can also be used by specifying an `xsl:use-attribute-sets` attribute on a literal result element. The value of the `xsl:use-attribute-sets` attribute is a whitespace-separated list of names of attribute sets. The `xsl:use-attribute-sets` attribute has the same effect as the `use-attribute-sets` attribute on `xsl:element` with the additional rule that attributes specified on the literal result element itself are treated as if they were specified by `xsl:attribute` elements before any actual `xsl:attribute` elements but after any `xsl:attribute` elements implied by the `xsl:use-attribute-sets` attribute. Thus, for a literal result element, attributes from attribute sets named in an `xsl:use-attribute-sets` attribute will be added first, in the order listed in the attribute; next, attributes specified on the literal result element will be added; finally, any attributes specified by `xsl:attribute` elements will be added. Since adding an attribute to an element replaces any existing attribute of that element with the same name, this means that attributes specified in attribute sets can be overridden by attributes specified on the literal result element itself.

The template within each `xsl:attribute` element in an `xsl:attribute-set` element is instantiated each time the

attribute set is used; it is instantiated using the same current node and current node list as is used for instantiating the element bearing the `use-attribute-sets` or `xsl:use-attribute-sets` attribute. However, it is the position in the stylesheet of the `xsl:attribute` element rather than of the element bearing the `use-attribute-sets` or `xsl:use-attribute-sets` attribute that determines which variable bindings are visible (see [11 Variables and Parameters]); thus, only variables and parameters declared by top-level `xsl:variable` and `xsl:param` elements are visible.

The following example creates a named attribute set `title-style` and uses it in a template rule.

```
<xsl:template match="chapter/heading">
  <fo:block quadding="start" xsl:use-attribute-sets="title-style">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:attribute-set name="title-style">
  <xsl:attribute name="font-size">12pt</xsl:attribute>
  <xsl:attribute name="font-weight">bold</xsl:attribute>
</xsl:attribute-set>
```

Multiple definitions of an attribute set with the same expanded-name are merged. An attribute from a definition that has higher import precedence takes precedence over an attribute from a definition that has lower import precedence. It is an error if there are two attribute sets that have the same expanded-name and equal import precedence and that both contain the same attribute, unless there is a definition of the attribute set with higher import precedence that also contains the attribute. An XSLT processor may signal the error; if it does not signal the error, it must recover by choosing from amongst the definitions that specify the attribute that have the highest import precedence the one that was specified last in the stylesheet. Where the attributes in an attribute set were specified is relevant only in merging the attributes into the attribute set; it makes no difference when the attribute set is used.

7.2 Creating Text

A template can also contain text nodes. Each text node in a template remaining after whitespace has been stripped as specified in [3.4 Whitespace Stripping] will create a text node with the same string-value in the result tree. Adjacent text nodes in the result tree are automatically merged.

Note that text is processed at the tree level. Thus, markup of `<` in a template will be represented in the stylesheet tree by a text node that includes the character `<`. This will create a text node in the result tree that contains a `<` character, which will be represented by the markup `<` (or an equivalent character reference) when the result tree is externalized as an XML document (unless output escaping is disabled as described in [16.4 Disabling Output Escaping]).

```
<!-- Category: instruction -->
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>
```

Literal data characters may also be wrapped in an `xsl:text` element. This wrapping may change what whitespace characters are stripped (see [3.4 Whitespace Stripping]) but does not affect how the characters are handled by the XSLT processor thereafter.

NOTE: The `xml:lang` and `xml:space` attributes are not treated specially by XSLT. In particular,

- it is the responsibility of the stylesheet author explicitly to generate any `xml:lang` or `xml:space` attributes that are needed in the result;
- specifying an `xml:lang` or `xml:space` attribute on an element in the XSLT namespace will not cause any `xml:lang` or `xml:space` attributes to appear in the result.

7.3 Creating Processing Instructions

```
<!-- Category: instruction -->
<xsl:processing-instruction
  name = { ncname }>
  <!-- Content: template -->
</xsl:processing-instruction>
```

The `xsl:processing-instruction` element is instantiated to create a processing instruction node. The content of

the `xsl:processing-instruction` element is a template for the string-value of the processing instruction node. The `xsl:processing-instruction` element has a required `name` attribute that specifies the name of the processing instruction node. The value of the `name` attribute is interpreted as an attribute value template.

For example, this

```
<xsl:processing-instruction name="xml-stylesheet">href="book.css" type="text/css"</xsl:processing-instructio
```

would create the processing instruction

```
<?xml-stylesheet href="book.css" type="text/css"?>
```

It is an error if the string that results from instantiating the `name` attribute is not both an NCName and a PITarget. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the processing instruction to the result tree.

> **NOTE:** This means that `xsl:processing-instruction` cannot be used to output an XML declaration. The `xsl:output` element should be used instead (see [16 Output]).

It is an error if instantiating the content of `xsl:processing-instruction` creates nodes other than text nodes. An XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the offending nodes together with their content.

It is an error if the result of instantiating the content of the `xsl:processing-instruction` contains the string `?>`. An XSLT processor may signal the error; if it does not signal the error, it must recover by inserting a space after any occurrence of `?` that is followed by a `>`.

7.4 Creating Comments

```
<!-- Category: instruction -->
<xsl:comment>
  <!-- Content: template -->
</xsl:comment>
```

The `xsl:comment` element is instantiated to create a comment node in the result tree. The content of the `xsl:comment` element is a template for the string-value of the comment node.

For example, this

```
<xsl:comment>This file is automatically generated. Do not edit!</xsl:comment>
```

would create the comment

```
<!--This file is automatically generated. Do not edit!-->
```

It is an error if instantiating the content of `xsl:comment` creates nodes other than text nodes. An XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the offending nodes together with their content.

It is an error if the result of instantiating the content of the `xsl:comment` contains the string `--` or ends with `-`. An XSLT processor may signal the error; if it does not signal the error, it must recover by inserting a space after any occurrence of `-` that is followed by another `-` or that ends the comment.

7.5 Copying

```
<!-- Category: instruction -->
<xsl:copy
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>
```

The `xsl:copy` element provides an easy way of copying the current node. Instantiating the `xsl:copy` element creates a copy of the current node. The namespace nodes of the current node are automatically copied as well, but the attributes and children of the node are not automatically copied. The content of the `xsl:copy` element is a template for the attributes and children of the created node; the content is instantiated only for nodes of types that can have attributes or children (i.e. root nodes and element nodes).

The `xsl:copy` element may have a `use-attribute-sets` attribute (see [7.1.4 Named Attribute Sets]). This

is used only when copying element nodes.

The root node is treated specially because the root node of the result tree is created implicitly. When the current node is the root node, `xsl:copy` will not create a root node, but will just use the content template.

For example, the identity transformation can be written using `xsl:copy` as follows:

```
<xsl:template match="@*|node()">
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

When the current node is an attribute, then if it would be an error to use `xsl:attribute` to create an attribute with the same name as the current node, then it is also an error to use `xsl:copy` (see [7.1.3 Creating Attributes with xsl:attribute]).

The following example shows how `xml:lang` attributes can be easily copied through from source to result. If a stylesheet defines the following named template:

```
<xsl:template name="apply-templates-copy-lang">
 <xsl:for-each select="@xml:lang">
   <xsl:copy/>
 </xsl:for-each>
 <xsl:apply-templates/>
</xsl:template>
```

then it can simply do

```
<xsl:call-template name="apply-templates-copy-lang"/>
```

instead of

```
<xsl:apply-templates/>
```

when it wants to copy the `xml:lang` attribute.

7.6 Computing Generated Text

Within a template, the `xsl:value-of` element can be used to compute generated text, for example by extracting text from the source tree or by inserting the value of a variable. The `xsl:value-of` element does this with an expression that is specified as the value of the `select` attribute. Expressions can also be used inside attribute values of literal result elements by enclosing the expression in curly braces (`{}`).

7.6.1 Generating Text with `xsl:value-of`

```
<!-- Category: instruction -->
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />
```

The `xsl:value-of` element is instantiated to create a text node in the result tree. The required `select` attribute is an expression; this expression is evaluated and the resulting object is converted to a string as if by a call to the **string** function. The string specifies the string-value of the created text node. If the string is empty, no text node will be created. The created text node will be merged with any adjacent text nodes.

The `xsl:copy-of` element can be used to copy a node-set over to the result tree without converting it to a string. See [11.3 Using Values of Variables and Parameters with xsl:copy-of].

For example, the following creates an HTML paragraph from a `person` element with `given-name` and `family-name` attributes. The paragraph will contain the value of the `given-name` attribute of the current node followed by a space and the value of the `family-name` attribute of the current node.

```
<xsl:template match="person">
  <p>
   <xsl:value-of select="@given-name"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="@family-name"/>
  </p>
</xsl:template>
```


For another example, the following creates an HTML paragraph from a `person` element with `given-name` and `family-name` children elements. The paragraph will contain the string-value of the first `given-name` child element of the current node followed by a space and the string-value of the first `family-name` child element of the current node.

```
<xsl:template match="person">
  <p>
   <xsl:value-of select="given-name"/>
   <xsl:text> </xsl:text>
   <xsl:value-of select="family-name"/>
  </p>
</xsl:template>
```

The following precedes each `procedure` element with a paragraph containing the security level of the procedure. It assumes that the security level that applies to a procedure is determined by a `security` attribute on the procedure element or on an ancestor element of the procedure. It also assumes that if more than one such element has a `security` attribute then the security level is determined by the element that is closest to the procedure.

```
<xsl:template match="procedure">
  <fo:block>
    <xsl:value-of select="ancestor-or-self::*[@security][1]/@security"/>
  </fo:block>
  <xsl:apply-templates/>
</xsl:template>
```

7.6.2 Attribute Value Templates

In an attribute value that is interpreted as an **attribute value template**, such as an attribute of a literal result element, an expression can be used by surrounding the expression with curly braces (`{}`). The attribute value template is instantiated by replacing the expression together with surrounding curly braces by the result of evaluating the expression and converting the resulting object to a string as if by a call to the **string** function. Curly braces are not recognized in an attribute value in an XSLT stylesheet unless the attribute is specifically stated to be one that is interpreted as an attribute value template; in an element syntax summary, the value of such attributes is surrounded by curly braces.

> **NOTE:** Not all attributes are interpreted as attribute value templates. Attributes whose value is an expression or pattern, attributes of top-level elements and attributes that refer to named XSLT objects are not interpreted as attribute value templates. In addition, `xmlns` attributes are not interpreted as attribute value templates; it would not be conformant with the XML Namespaces Recommendation to do this.

The following example creates an `img` result element from a `photograph` element in the source; the value of the `src` attribute of the `img` element is computed from the value of the `image-dir` variable and the string-value of the `href` child of the `photograph` element; the value of the `width` attribute of the `img` element is computed from the value of the `width` attribute of the `size` child of the `photograph` element:

```
<xsl:variable name="image-dir">/images</xsl:variable>

<xsl:template match="photograph">
<img src="{$image-dir}/{href}" width="{size/@width}"/>
</xsl:template>
```

With this source

```
<photograph>
  <href>headquarters.jpg</href>
  <size width="300"/>
</photograph>
```

the result would be

```
<img src="/images/headquarters.jpg" width="300"/>
```

When an attribute value template is instantiated, a double left or right curly brace outside an expression will be replaced by a single curly brace. It is an error if a right curly brace occurs in an attribute value template outside an expression without being followed by a second right curly brace. A right curly brace inside a Literal in an expression is not recognized as terminating the expression.

Curly braces are *not* recognized recursively inside expressions. For example:

```
<a href="#{id({@ref})/title}">
```

is *not* allowed. Instead, use simply:

```
<a href="#{id(@ref)/title}">
```

7.7 Numbering

```
<!-- Category: instruction -->
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />
```

The `xsl:number` element is used to insert a formatted number into the result tree. The number to be inserted may be specified by an expression. The `value` attribute contains an expression. The expression is evaluated and the resulting object is converted to a number as if by a call to the **number** function. The number is rounded to an integer and then converted to a string using the attributes specified in **[7.7.1 Number to String Conversion Attributes]**; in this context, the value of each of these attributes is interpreted as an attribute value template. After conversion, the resulting string is inserted in the result tree. For example, the following example numbers a sorted list:

```
<xsl:template match="items">
  <xsl:for-each select="item">
    <xsl:sort select="."/>
    <p>
      <xsl:number value="position()" format="1. "/>
      <xsl:value-of select="."/>
    </p>
  </xsl:for-each>
</xsl:template>
```

If no `value` attribute is specified, then the `xsl:number` element inserts a number based on the position of the current node in the source tree. The following attributes control how the current node is to be numbered:

- The `level` attribute specifies what levels of the source tree should be considered; it has the values `single`, `multiple` or `any`. The default is `single`.
- The `count` attribute is a pattern that specifies what nodes should be counted at those levels. If `count` attribute is not specified, then it defaults to the pattern that matches any node with the same node type as the current node and, if the current node has an expanded-name, with the same expanded-name as the current node.
- The `from` attribute is a pattern that specifies where counting starts.

In addition, the attributes specified in **[7.7.1 Number to String Conversion Attributes]** are used for number to string conversion, as in the case when the `value` attribute is specified.

The `xsl:number` element first constructs a list of positive integers using the `level`, `count` and `from` attributes:

- When `level="single"`, it goes up to the first node in the ancestor-or-self axis that matches the `count` pattern, and constructs a list of length one containing one plus the number of preceding siblings of that ancestor that match the `count` pattern. If there is no such ancestor, it constructs an empty list. If the `from` attribute is specified, then the only ancestors that are searched are those that are descendants of the nearest ancestor that matches the `from` pattern. Preceding siblings has the same meaning here as with the `preceding-sibling` axis.
- When `level="multiple"`, it constructs a list of all ancestors of the current node in document order followed by the element itself; it then selects from the list those nodes that match the `count` pattern; it then maps each node in the list to one plus the number of preceding siblings of that node that match the `count` pattern. If the `from` attribute is specified, then the only ancestors that are searched are those that are descendants of the nearest ancestor that matches the `from` pattern. Preceding siblings has the same meaning here as with the `preceding-sibling` axis.

- When `level="any"`, it constructs a list of length one containing the number of nodes that match the `count` pattern and belong to the set containing the current node and all nodes at any level of the document that are before the current node in document order, excluding any namespace and attribute nodes (in other words the union of the members of the `preceding` and `ancestor-or-self` axes). If the `from` attribute is specified, then only nodes after the first node before the current node that match the `from` pattern are considered.

The list of numbers is then converted into a string using the attributes specified in **[7.7.1 Number to String Conversion Attributes]**; in this context, the value of each of these attributes is interpreted as an attribute value template. After conversion, the resulting string is inserted in the result tree.

The following would number the items in an ordered list:

```
<xsl:template match="ol/item">
  <fo:block>
    <xsl:number/><xsl:text>. </xsl:text><xsl:apply-templates/>
  </fo:block>
<xsl:template>
```

The following two rules would number `title` elements. This is intended for a document that contains a sequence of chapters followed by a sequence of appendices, where both chapters and appendices contain sections, which in turn contain subsections. Chapters are numbered 1, 2, 3; appendices are numbered A, B, C; sections in chapters are numbered 1.1, 1.2, 1.3; sections in appendices are numbered A.1, A.2, A.3.

```
<xsl:template match="title">
  <fo:block>
     <xsl:number level="multiple"
                 count="chapter|section|subsection"
                 format="1.1 "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="appendix//title" priority="1">
  <fo:block>
     <xsl:number level="multiple"
                 count="appendix|section|subsection"
                 format="A.1 "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The following example numbers notes sequentially within a chapter:

```
<xsl:template match="note">
  <fo:block>
     <xsl:number level="any" from="chapter" format="(1) "/>
     <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The following example would number `H4` elements in HTML with a three-part label:

```
<xsl:template match="H4">
 <fo:block>
   <xsl:number level="any" from="H1" count="H2"/>
   <xsl:text>.</xsl:text>
   <xsl:number level="any" from="H2" count="H3"/>
   <xsl:text>.</xsl:text>
   <xsl:number level="any" from="H3" count="H4"/>
   <xsl:text> </xsl:text>
   <xsl:apply-templates/>
 </fo:block>
</xsl:template>
```

7.7.1 Number to String Conversion Attributes

The following attributes are used to control conversion of a list of numbers into a string. The numbers are integers greater than 0. The attributes are all optional.

The main attribute is `format`. The default value for the `format` attribute is `1`. The `format` attribute is split into a sequence of tokens where each token is a maximal sequence of alphanumeric characters or a maximal sequence of non-alphanumeric characters. Alphanumeric means any character that has a Unicode category of Nd, Nl, No, Lu, Ll, Lt, Lm or Lo. The alphanumeric tokens (format tokens) specify the format to be used for each number in the list. If the first token is a non-alphanumeric token, then the constructed string will start with that token; if the last token is non-alphanumeric token, then the constructed string will end with that

token. Non-alphanumeric tokens that occur between two format tokens are separator tokens that are used to join numbers in the list. The *n*th format token will be used to format the *n*th number in the list. If there are more numbers than format tokens, then the last format token will be used to format remaining numbers. If there are no format tokens, then a format token of `1` is used to format all numbers. The format token specifies the string to be used to represent the number 1. Each number after the first will be separated from the preceding number by the separator token preceding the format token used to format that number, or, if there are no separator tokens, then by `.` (a period character).

Format tokens are a superset of the allowed values for the `type` attribute for the `OL` element in HTML 4.0 and are interpreted as follows:

- Any token where the last character has a decimal digit value of 1 (as specified in the Unicode character property database), and the Unicode value of preceding characters is one less than the Unicode value of the last character generates a decimal representation of the number where each number is at least as long as the format token. Thus, a format token `1` generates the sequence `1 2 ... 10 11 12 ...`, and a format token `01` generates the sequence `01 02 ... 09 10 11 12 ... 99 100 101`.
- A format token `A` generates the sequence `A B C ... Z AA AB AC...`.
- A format token `a` generates the sequence `a b c ... z aa ab ac...`.
- A format token `i` generates the sequence `i ii iii iv v vi vii viii ix x ...`.
- A format token `I` generates the sequence `I II III IV V VI VII VIII IX X ...`.
- Any other format token indicates a numbering sequence that starts with that token. If an implementation does not support a numbering sequence that starts with that token, it must use a format token of `1`.

When numbering with an alphabetic sequence, the `lang` attribute specifies which language's alphabet is to be used; it has the same range of values as `xml:lang` [XML]; if no `lang` value is specified, the language should be determined from the system environment. Implementers should document for which languages they support numbering.

> **NOTE:**Implementers should not make any assumptions about how numbering works in particular languages and should properly research the languages that they wish to support. The numbering conventions of many languages are very different from English.

The `letter-value` attribute disambiguates between numbering sequences that use letters. In many languages there are two commonly used numbering sequences that use letters. One numbering sequence assigns numeric values to letters in alphabetic sequence, and the other assigns numeric values to each letter in some other manner traditional in that language. In English, these would correspond to the numbering sequences specified by the format tokens `a` and `i`. In some languages, the first member of each sequence is the same, and so the format token alone would be ambiguous. A value of `alphabetic` specifies the alphabetic sequence; a value of `traditional` specifies the other sequence. If the `letter-value` attribute is not specified, then it is implementation-dependent how any ambiguity is resolved.

> **NOTE:**It is possible for two conforming XSLT processors not to convert a number to exactly the same string. Some XSLT processors may not support some languages. Furthermore, there may be variations possible in the way conversions are performed for any particular language that are not specifiable by the attributes on `xsl:number`. Future versions of XSLT may provide additional attributes to provide control over these variations. Implementations may also use implementation-specific namespaced attributes on `xsl:number` for this.

The `grouping-separator` attribute gives the separator used as a grouping (e.g. thousands) separator in decimal numbering sequences, and the optional `grouping-size` specifies the size (normally 3) of the grouping. For example, `grouping-separator=","` and `grouping-size="3"` would produce numbers of the form `1,000,000`. If only one of the `grouping-separator` and `grouping-size` attributes is specified, then it is ignored.

Here are some examples of conversion specifications:

- `format="&#x30A2;"` specifies Katakana numbering

- `format="&#x30A4;"` specifies Katakana numbering in the "iroha" order
- `format="&#x0E51;"` specifies numbering with Thai digits
- `format="&#x05D0;" letter-value="traditional"` specifies "traditional" Hebrew numbering
- `format="&#x10D0;" letter-value="traditional"` specifies Georgian numbering
- `format="&#x03B1;" letter-value="traditional"` specifies "classical" Greek numbering
- `format="&#x0430;" letter-value="traditional"` specifies Old Slavic numbering

8 Repetition

```
<!-- Category: instruction -->
<xsl:for-each
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>
```

When the result has a known regular structure, it is useful to be able to specify directly the template for selected nodes. The `xsl:for-each` instruction contains a template, which is instantiated for each node selected by the expression specified by the `select` attribute. The `select` attribute is required. The expression must evaluate to a node-set. The template is instantiated with the selected node as the current node, and with a list of all of the selected nodes as the current node list. The nodes are processed in document order, unless a sorting specification is present (see [10 Sorting]).

For example, given an XML document with this structure

```
<customers>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
  <customer>
    <name>...</name>
    <order>...</order>
    <order>...</order>
  </customer>
</customers>
```

the following would create an HTML document containing a table with a row for each `customer` element

```
<xsl:template match="/">
  <html>
    <head>
      <title>Customers</title>
    </head>
    <body>
      <table>
        <tbody>
          <xsl:for-each select="customers/customer">
            <tr>
              <th>
                <xsl:apply-templates select="name"/>
              </th>
              <xsl:for-each select="order">
                <td>
                  <xsl:apply-templates/>
                </td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </tbody>
      </table>
    </body>
  </html>
</xsl:template>
```

9 Conditional Processing

There are two instructions in XSLT that support conditional processing in a template: `xsl:if` and `xsl:choose`. The `xsl:if` instruction provides simple if-then conditionality; the `xsl:choose` instruction supports selection of one choice when there are several possibilities.

9.1 Conditional Processing with `xsl:if`

```
<!-- Category: instruction -->
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>
```

The xsl:if element has a test attribute, which specifies an expression. The content is a template. The expression is evaluated and the resulting object is converted to a boolean as if by a call to the **boolean** function. If the result is true, then the content template is instantiated; otherwise, nothing is created. In the following example, the names in a group of names are formatted as a comma separated list:

```
<xsl:template match="namelist/name">
  <xsl:apply-templates/>
  <xsl:if test="not(position()=last())">, </xsl:if>
</xsl:template>
```

The following colors every other table row yellow:

```
<xsl:template match="item">
  <tr>
    <xsl:if test="position() mod 2 = 0">
       <xsl:attribute name="bgcolor">yellow</xsl:attribute>
    </xsl:if>
    <xsl:apply-templates/>
  </tr>
</xsl:template>
```

9.2 Conditional Processing with xsl:choose

```
<!-- Category: instruction -->
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
```

```
<xsl:when
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>
```

```
<xsl:otherwise>
  <!-- Content: template -->
</xsl:otherwise>
```

The xsl:choose element selects one among a number of possible alternatives. It consists of a sequence of xsl:when elements followed by an optional xsl:otherwise element. Each xsl:when element has a single attribute, test, which specifies an expression. The content of the xsl:when and xsl:otherwise elements is a template. When an xsl:choose element is processed, each of the xsl:when elements is tested in turn, by evaluating the expression and converting the resulting object to a boolean as if by a call to the **boolean** function. The content of the first, and only the first, xsl:when element whose test is true is instantiated. If no xsl:when is true, the content of the xsl:otherwise element is instantiated. If no xsl:when element is true, and no xsl:otherwise element is present, nothing is created.

The following example enumerates items in an ordered list using arabic numerals, letters, or roman numerals depending on the depth to which the ordered lists are nested.

```
<xsl:template match="orderedlist/listitem">
  <fo:list-item indent-start='2pi'>
    <fo:list-item-label>
      <xsl:variable name="level"
                    select="count(ancestor::orderedlist) mod 3"/>
      <xsl:choose>
        <xsl:when test='$level=1'>
          <xsl:number format="i"/>
        </xsl:when>
        <xsl:when test='$level=2'>
          <xsl:number format="a"/>
        </xsl:when>
        <xsl:otherwise>
          <xsl:number format="1"/>
        </xsl:otherwise>
      </xsl:choose>
      <xsl:text>. </xsl:text>
    </fo:list-item-label>
    <fo:list-item-body>
      <xsl:apply-templates/>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>
```


10 Sorting

```
<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" | qname-but-not-ncname }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />
```

Sorting is specified by adding xsl:sort elements as children of an xsl:apply-templates or xsl:for-each element. The first xsl:sort child specifies the primary sort key, the second xsl:sort child specifies the secondary sort key and so on. When an xsl:apply-templates or xsl:for-each element has one or more xsl:sort children, then instead of processing the selected nodes in document order, it sorts the nodes according to the specified sort keys and then processes them in sorted order. When used in xsl:for-each, xsl:sort elements must occur first. When a template is instantiated by xsl:apply-templates and xsl:for-each, the current node list list consists of the complete list of nodes being processed in sorted order.

xsl:sort has a select attribute whose value is an expression. For each node to be processed, the expression is evaluated with that node as the current node and with the complete list of nodes being processed in unsorted order as the current node list. The resulting object is converted to a string as if by a call to the **string** function; this string is used as the sort key for that node. The default value of the select attribute is ., which will cause the string-value of the current node to be used as the sort key.

This string serves as a sort key for the node. The following optional attributes on xsl:sort control how the list of sort keys are sorted; the values of all of these attributes are interpreted as attribute value templates.

- order specifies whether the strings should be sorted in ascending or descending order; ascending specifies ascending order; descending specifies descending order; the default is ascending
- lang specifies the language of the sort keys; it has the same range of values as xml:lang [XML]; if no lang value is specified, the language should be determined from the system environment
- data-type specifies the data type of the strings; the following values are allowed:
  - text specifies that the sort keys should be sorted lexicographically in the culturally correct manner for the language specified by lang
  - number specifies that the sort keys should be converted to numbers and then sorted according to the numeric value; the sort key is converted to a number as if by a call to the **number** function; the lang attribute is ignored
  - a QName with a prefix is expanded into an expanded-name as described in [**2.4 Qualified Names**]; the expanded-name identifies the data-type; the behavior in this case is not specified by this document

  The default value is text.

  NOTE: The XSL Working Group plans that future versions of XSLT will leverage XML Schemas to define further values for this attribute.
- case-order has the value upper-first or lower-first; this applies when data-type="text", and specifies that upper-case letters should sort before lower-case letters or vice-versa respectively. For example, if lang="en", then A a B b are sorted with case-order="upper-first" and a A b B are sorted with case-order="lower-first". The default value is language dependent.

  NOTE: It is possible for two conforming XSLT processors not to sort exactly the same. Some XSLT processors may not support some languages. Furthermore, there may be variations possible in the sorting of any particular language that are not specified by the attributes on xsl:sort, for example, whether Hiragana or Katakana is sorted first in Japanese. Future versions of XSLT may provide additional attributes to provide control over these variations. Implementations may also use implementation-specific namespaced attributes on xsl:sort for this.

  NOTE: It is recommended that implementers consult [UNICODE TR10] for information on

internationalized sorting.

The sort must be stable: in the sorted list of nodes, any sub list that has sort keys that all compare equal must be in document order.

For example, suppose an employee database has the form

```
<employees>
  <employee>
    <name>
      <given>James</given>
      <family>Clark</family>
    </name>
    ...
  </employee>
</employees>
```

Then a list of employees sorted by name could be generated using:

```
<xsl:template match="employees">
  <ul>
    <xsl:apply-templates select="employee">
      <xsl:sort select="name/family"/>
      <xsl:sort select="name/given"/>
    </xsl:apply-templates>
  </ul>
</xsl:template>

<xsl:template match="employee">
  <li>
    <xsl:value-of select="name/given"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="name/family"/>
  </li>
</xsl:template>
```

11 Variables and Parameters

```
<!-- Category: top-level-element -->
<!-- Category: instruction -->
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>
```

```
<!-- Category: top-level-element -->
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>
```

A variable is a name that may be bound to a value. The value to which a variable is bound (the **value** of the variable) can be an object of any of the types that can be returned by expressions. There are two elements that can be used to bind variables: `xsl:variable` and `xsl:param`. The difference is that the value specified on the `xsl:param` variable is only a default value for the binding; when the template or stylesheet within which the `xsl:param` element occurs is invoked, parameters may be passed that are used in place of the default values.

Both `xsl:variable` and `xsl:param` have a required `name` attribute, which specifies the name of the variable. The value of the `name` attribute is a QName, which is expanded as described in **[2.4 Qualified Names]**.

For any use of these variable-binding elements, there is a region of the stylesheet tree within which the binding is visible; within this region, any binding of the variable that was visible on the variable-binding element itself is hidden. Thus, only the innermost binding of a variable is visible. The set of variable bindings in scope for an expression consists of those bindings that are visible at the point in the stylesheet where the expression occurs.

11.1 Result Tree Fragments

Variables introduce an additional data-type into the expression language. This additional data type is called **result tree fragment**. A variable may be bound to a result tree fragment instead of one of the four basic XPath data-types (string, number, boolean, node-set). A result tree fragment represents a fragment of the result tree. A result tree fragment is treated equivalently to a node-set that contains just a single root node. However, the operations permitted on a result tree fragment are a subset of those permitted on a node-set.

An operation is permitted on a result tree fragment only if that operation would be permitted on a string (the operation on the string may involve first converting the string to a number or boolean). In particular, it is not permitted to use the `/`, `//`, and `[]` operators on result tree fragments. When a permitted operation is performed on a result tree fragment, it is performed exactly as it would be on the equivalent node-set.

When a result tree fragment is copied into the result tree (see **[11.3 Using Values of Variables and Parameters with `xsl:copy-of`]**), then all the nodes that are children of the root node in the equivalent node-set are added in sequence to the result tree.

Expressions can only return values of type result tree fragment by referencing variables of type result tree fragment or calling extension functions that return a result tree fragment or getting a system property whose value is a result tree fragment.

11.2 Values of Variables and Parameters

A variable-binding element can specify the value of the variable in three alternative ways.

- If the variable-binding element has a `select` attribute, then the value of the attribute must be an expression and the value of the variable is the object that results from evaluating the expression. In this case, the content must be empty.

- If the variable-binding element does not have a `select` attribute and has non-empty content (i.e. the variable-binding element has one or more child nodes), then the content of the variable-binding element specifies the value. The content of the variable-binding element is a template, which is instantiated to give the value of the variable. The value is a result tree fragment equivalent to a node-set containing just a single root node having as children the sequence of nodes produced by instantiating the template. The base URI of the nodes in the result tree fragment is the base URI of the variable-binding element.

  It is an error if a member of the sequence of nodes created by instantiating the template is an attribute node or a namespace node, since a root node cannot have an attribute node or a namespace node as a child. An XSLT processor may signal the error; if it does not signal the error, it must recover by not adding the attribute node or namespace node.

- If the variable-binding element has empty content and does not have a `select` attribute, then the value of the variable is an empty string. Thus

  ```
  <xsl:variable name="x"/>
  ```

  is equivalent to

  ```
  <xsl:variable name="x" select="''"/>
  ```

  NOTE: When a variable is used to select nodes by position, be careful not to do:

  ```
  <xsl:variable name="n">2</xsl:variable>
  ...
  <xsl:value-of select="item[$n]"/>
  ```

  This will output the value of the first item element, because the variable `n` will be bound to a result tree fragment, not a number. Instead, do either

  ```
  <xsl:variable name="n" select="2"/>
  ...
  <xsl:value-of select="item[$n]"/>
  ```

  or

  ```
  <xsl:variable name="n">2</xsl:variable>
  ...
  <xsl:value-of select="item[position()=$n]"/>
  ```

  NOTE: One convenient way to specify the empty node-set as the default value of a parameter is:

  ```
  <xsl:param name="x" select="/.."/>
  ```

11.3 Using Values of Variables and Parameters with `xsl:copy-of`

```
<!-- Category: instruction -->
```

```
<xsl:copy-of
  select = expression />
```

The xsl:copy-of element can be used to insert a result tree fragment into the result tree, without first converting it to a string as xsl:value-of does (see [7.6.1 Generating Text with xsl:value-of]). The required select attribute contains an expression. When the result of evaluating the expression is a result tree fragment, the complete fragment is copied into the result tree. When the result is a node-set, all the nodes in the set are copied in document order into the result tree; copying an element node copies the attribute nodes, namespace nodes and children of the element node as well as the element node itself; a root node is copied by copying its children. When the result is neither a node-set nor a result tree fragment, the result is converted to a string and then inserted into the result tree, as with xsl:value-of.

11.4 Top-level Variables and Parameters

Both xsl:variable and xsl:param are allowed as top-level elements. A top-level variable-binding element declares a global variable that is visible everywhere. A top-level xsl:param element declares a parameter to the stylesheet; XSLT does not define the mechanism by which parameters are passed to the stylesheet. It is an error if a stylesheet contains more than one binding of a top-level variable with the same name and same import precedence. At the top-level, the expression or template specifying the variable value is evaluated with the same context as that used to process the root node of the source document: the current node is the root node of the source document and the current node list is a list containing just the root node of the source document. If the template or expression specifying the value of a global variable *x* references a global variable *y*, then the value for *y* must be computed before the value of *x*. It is an error if it is impossible to do this for all global variable definitions; in other words, it is an error if the definitions are circular.

This example declares a global variable para-font-size, which it references in an attribute value template.

```
<xsl:variable name="para-font-size">12pt</xsl:variable>

<xsl:template match="para">
 <fo:block font-size="{$para-font-size}">
   <xsl:apply-templates/>
 </fo:block>
</xsl:template>
```

11.5 Variables and Parameters within Templates

As well as being allowed at the top-level, both xsl:variable and xsl:param are also allowed in templates. xsl:variable is allowed anywhere within a template that an instruction is allowed. In this case, the binding is visible for all following siblings and their descendants. Note that the binding is not visible for the xsl:variable element itself. xsl:param is allowed as a child at the beginning of an xsl:template element. In this context, the binding is visible for all following siblings and their descendants. Note that the binding is not visible for the xsl:param element itself.

A binding **shadows** another binding if the binding occurs at a point where the other binding is visible, and the bindings have the same name. It is an error if a binding established by an xsl:variable or xsl:param element within a template shadows another binding established by an xsl:variable or xsl:param element also within the template. It is not an error if a binding established by an xsl:variable or xsl:param element in a template shadows another binding established by an xsl:variable or xsl:param top-level element. Thus, the following is an error:

```
<xsl:template name="foo">
<xsl:param name="x" select="1"/>
<xsl:variable name="x" select="2"/>
</xsl:template>
```

However, the following is allowed:

```
<xsl:param name="x" select="1"/>
<xsl:template name="foo">
<xsl:variable name="x" select="2"/>
</xsl:template>
```

NOTE: The nearest equivalent in Java to an xsl:variable element in a template is a final local variable declaration with an initializer. For example,

```
<xsl:variable name="x" select="'value'"/>
```

has similar semantics to


```
final Object x = "value";
```

XSLT does not provide an equivalent to the Java assignment operator

```
x = "value";
```

because this would make it harder to create an implementation that processes a document other than in a batch-like way, starting at the beginning and continuing through to the end.

11.6 Passing Parameters to Templates

```
<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

Parameters are passed to templates using the xsl:with-param element. The required name attribute specifies the name of the parameter (the variable the value of whose binding is to be replaced). The value of the name attribute is a QName, which is expanded as described in [2.4 Qualified Names]. xsl:with-param is allowed within both xsl:call-template and xsl:apply-templates. The value of the parameter is specified in the same way as for xsl:variable and xsl:param. The current node and current node list used for computing the value specified by xsl:with-param element is the same as that used for the xsl:apply-templates or xsl:call-template element within which it occurs. It is not an error to pass a parameter *x* to a template that does not have an xsl:param element for *x*; the parameter is simply ignored.

This example defines a named template for a numbered-block with an argument to control the format of the number.

```
<xsl:template name="numbered-block">
  <xsl:param name="format">1. </xsl:param>
  <fo:block>
    <xsl:number format="{$format}"/>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="ol//ol/li">
  <xsl:call-template name="numbered-block">
    <xsl:with-param name="format">a. </xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

12 Additional Functions

This section describes XSLT-specific additions to the core XPath function library. Some of these additional functions also make use of information specified by top-level elements in the stylesheet; this section also describes these elements.

12.1 Multiple Source Documents

Function: *node-set* **document**(*object*, *node-set*?)

The **document** function allows access to XML documents other than the main source document.

When the **document** function has exactly one argument and the argument is a node-set, then the result is the union, for each node in the argument node-set, of the result of calling the **document** function with the first argument being the string-value of the node, and the second argument being a node-set with the node as its only member. When the **document** function has two arguments and the first argument is a node-set, then the result is the union, for each node in the argument node-set, of the result of calling the **document** function with the first argument being the string-value of the node, and with the second argument being the second argument passed to the **document** function.

When the first argument to the **document** function is not a node-set, the first argument is converted to a string as if by a call to the **string** function. This string is treated as a URI reference; the resource identified by the URI is retrieved. The data resulting from the retrieval action is parsed as an XML document and a tree is constructed in accordance with the data model (see [3 Data Model]). If there is an error retrieving the resource, then the XSLT processor may signal an error; if it does not signal an error, it must recover by returning an empty node-set. One possible kind of retrieval error is that the XSLT processor does not support

the URI scheme used by the URI. An XSLT processor is not required to support any particular URI schemes. The documentation for an XSLT processor should specify which URI schemes the XSLT processor supports.

If the URI reference does not contain a fragment identifier, then a node-set containing just the root node of the document is returned. If the URI reference does contain a fragment identifier, the function returns a node-set containing the nodes in the tree identified by the fragment identifier of the URI reference. The semantics of the fragment identifier is dependent on the media type of the result of retrieving the URI. If there is an error in processing the fragment identifier, the XSLT processor may signal the error; if it does not signal the error, it must recover by returning an empty node-set. Possible errors include:

- The fragment identifier identifies something that cannot be represented by an XSLT node-set (such as a range of characters within a text node).
- The XSLT processor does not support fragment identifiers for the media-type of the retrieval result. An XSLT processor is not required to support any particular media types. The documentation for an XSLT processor should specify for which media types the XSLT processor supports fragment identifiers.

The data resulting from the retrieval action is parsed as an XML document regardless of the media type of the retrieval result; if the top-level media type is `text`, then it is parsed in the same way as if the media type were `text/xml`; otherwise, it is parsed in the same way as if the media type were `application/xml`.

> **NOTE:**Since there is no top-level `xml` media type, data with a media type other than `text/xml` or `application/xml` may in fact be XML.

The URI reference may be relative. The base URI (see [3.2 Base URI]) of the node in the second argument node-set that is first in document order is used as the base URI for resolving the relative URI into an absolute URI. If the second argument is omitted, then it defaults to the node in the stylesheet that contains the expression that includes the call to the **document** function. Note that a zero-length URI reference is a reference to the document relative to which the URI reference is being resolved; thus `document("")` refers to the root node of the stylesheet; the tree representation of the stylesheet is exactly the same as if the XML document containing the stylesheet was the initial source document.

Two documents are treated as the same document if they are identified by the same URI. The URI used for the comparison is the absolute URI into which any relative URI was resolved and does not include any fragment identifier. One root node is treated as the same node as another root node if the two nodes are from the same document. Thus, the following expression will always be true:

```
generate-id(document("foo.xml"))=generate-id(document("foo.xml"))
```

The **document** function gives rise to the possibility that a node-set may contain nodes from more than one document. With such a node-set, the relative document order of two nodes in the same document is the normal document order defined by XPath [XPath]. The relative document order of two nodes in different documents is determined by an implementation-dependent ordering of the documents containing the two nodes. There are no constraints on how the implementation orders documents other than that it must do so consistently: an implementation must always use the same order for the same set of documents.

12.2 Keys

Keys provide a way to work with documents that contain an implicit cross-reference structure. The `ID`, `IDREF` and `IDREFS` attribute types in XML provide a mechanism to allow XML documents to make their cross-reference explicit. XSLT supports this through the XPath **id** function. However, this mechanism has a number of limitations:

- ID attributes must be declared as such in the DTD. If an ID attribute is declared as an ID attribute only in the external DTD subset, then it will be recognized as an ID attribute only if the XML processor reads the external DTD subset. However, XML does not require XML processors to read the external DTD, and they may well choose not to do so, especially if the document is declared `standalone="yes"`.
- A document can contain only a single set of unique IDs. There cannot be separate independent sets of unique IDs.
- The ID of an element can only be specified in an attribute; it cannot be specified by the content of the element, or by a child element.

- An ID is constrained to be an XML name. For example, it cannot contain spaces.
- An element can have at most one ID.
- At most one element can have a particular ID.

Because of these limitations XML documents sometimes contain a cross-reference structure that is not explicitly declared by ID/IDREF/IDREFS attributes.

A key is a triple containing:

1. the node which has the key
2. the name of the key (an expanded-name)
3. the value of the key (a string)

A stylesheet declares a set of keys for each document using the `xsl:key` element. When this set of keys contains a member with node *x*, name *y* and value *z*, we say that node *x* has a key with name *y* and value *z*.

Thus, a key is a kind of generalized ID, which is not subject to the same limitations as an XML ID:

- Keys are declared in the stylesheet using `xsl:key` elements.
- A key has a name as well as a value; each key name may be thought of as distinguishing a separate, independent space of identifiers.
- The value of a named key for an element may be specified in any convenient place; for example, in an attribute, in a child element or in content. An XPath expression is used to specify where to find the value for a particular named key.
- The value of a key can be an arbitrary string; it is not constrained to be a name.
- There can be multiple keys in a document with the same node, same key name, but different key values.
- There can be multiple keys in a document with the same key name, same key value, but different nodes.

```
<!-- Category: top-level-element -->
<xsl:key
  name = gname
  match = pattern
  use = expression />
```

The `xsl:key` element is used to declare keys. The `name` attribute specifies the name of the key. The value of the `name` attribute is a QName, which is expanded as described in [2.4 Qualified Names]. The `match` attribute is a Pattern; an `xsl:key` element gives information about the keys of any node that matches the pattern specified in the match attribute. The `use` attribute is an expression specifying the values of the key; the expression is evaluated once for each node that matches the pattern. If the result is a node-set, then for each node in the node-set, the node that matches the pattern has a key of the specified name whose value is the string-value of the node in the node-set; otherwise, the result is converted to a string, and the node that matches the pattern has a key of the specified name with value equal to that string. Thus, a node *x* has a key with name *y* and value *z* if and only if there is an `xsl:key` element such that:

- *x* matches the pattern specified in the `match` attribute of the `xsl:key` element;
- the value of the `name` attribute of the `xsl:key` element is equal to *y*; and
- when the expression specified in the `use` attribute of the `xsl:key` element is evaluated with *x* as the current node and with a node list containing just *x* as the current node list resulting in an object *u*, then either *z* is equal to the result of converting *u* to a string as if by a call to the **string** function, or *u* is a node-set and *z* is equal to the string-value of one or more of the nodes in *u*.

Note also that there may be more than one `xsl:key` element that matches a given node; all of the matching `xsl:key` elements are used, even if they do not have the same import precedence.

It is an error for the value of either the `use` attribute or the `match` attribute to contain a VariableReference.

Function: *node-set* **key**(*string*, *object*)

The **key** function does for keys what the **id** function does for IDs. The first argument specifies the name of the key. The value of the argument must be a QName, which is expanded as described in **[2.4 Qualified Names]**. When the second argument to the **key** function is of type node-set, then the result is the union of the result of applying the **key** function to the string value of each of the nodes in the argument node-set. When the second argument to **key** is of any other type, the argument is converted to a string as if by a call to the **string** function; it returns a node-set containing the nodes in the same document as the context node that have a value for the named key equal to this string.

For example, given a declaration

```
<xsl:key name="idkey" match="div" use="@id"/>
```

an expression `key("idkey",@ref)` will return the same node-set as `id(@ref)`, assuming that the only ID attribute declared in the XML source document is:

```
<!ATTLIST div id ID #IMPLIED>
```

and that the `ref` attribute of the current node contains no whitespace.

Suppose a document describing a function library uses a `prototype` element to define functions

```
<prototype name="key" return-type="node-set">
<arg type="string"/>
<arg type="object"/>
</prototype>
```

and a `function` element to refer to function names

```
<function>key</function>
```

Then the stylesheet could generate hyperlinks between the references and definitions as follows:

```
<xsl:key name="func" match="prototype" use="@name"/>

<xsl:template match="function">
<b>
  <a href="#{generate-id(key('func',.))}">
    <xsl:apply-templates/>
  </a>
</b>
</xsl:template>

<xsl:template match="prototype">
<p><a name="{generate-id()}">
<b>Function: </b>
...
</a></p>
</xsl:template>
```

The **key** can be used to retrieve a key from a document other than the document containing the context node. For example, suppose a document contains bibliographic references in the form `<bibref>XSLT</bibref>`, and there is a separate XML document `bib.xml` containing a bibliographic database with entries in the form:

```
<entry name="XSLT">...</entry>
```

Then the stylesheet could use the following to transform the `bibref` elements:

```
<xsl:key name="bib" match="entry" use="@name"/>

<xsl:template match="bibref">
  <xsl:variable name="name" select="."/>
  <xsl:for-each select="document('bib.xml')">
    <xsl:apply-templates select="key('bib',$name)"/>
  </xsl:for-each>
</xsl:template>
```

12.3 Number Formatting

Function: *string* **format-number**(*number*, *string*, *string*?)

The **format-number** function converts its first argument to a string using the format pattern string specified by the second argument and the decimal-format named by the third argument, or the default decimal-format, if there is no third argument. The format pattern string is in the syntax specified by the JDK 1.1 DecimalFormat class. The format pattern string is in a localized notation: the decimal-format determines what characters have a special meaning in the pattern (with the exception of the quote character, which is not localized). The format pattern must not contain the currency sign (#x00A4); support for this feature was added after the initial release of JDK 1.1. The decimal-format name must be a QName, which is expanded as described in **[2.4 Qualified Names]**. It is an error if the stylesheet does not contain a declaration of the decimal-format with the specified expanded-name.

> **NOTE:**Implementations are not required to use the JDK 1.1 implementation, nor are implementations required to be implemented in Java.

> **NOTE:**Stylesheets can use other facilities in XPath to control rounding.

```
<!-- Category: top-level-element -->
<xsl:decimal-format
  name = qname
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />
```

The `xsl:decimal-format` element declares a decimal-format, which controls the interpretation of a format pattern used by the **format-number** function. If there is a `name` attribute, then the element declares a named decimal-format; otherwise, it declares the default decimal-format. The value of the `name` attribute is a QName, which is expanded as described in **[2.4 Qualified Names]**. It is an error to declare either the default decimal-format or a decimal-format with a given name more than once (even with different import precedence), unless it is declared every time with the same value for all attributes (taking into account any default values).

The other attributes on `xsl:decimal-format` correspond to the methods on the JDK 1.1 DecimalFormatSymbols class. For each `get`/`set` method pair there is an attribute defined for the `xsl:decimal-format` element.

The following attributes both control the interpretation of characters in the format pattern and specify characters that may appear in the result of formatting the number:

- `decimal-separator` specifies the character used for the decimal sign; the default value is the period character (`.`)
- `grouping-separator` specifies the character used as a grouping (e.g. thousands) separator; the default value is the comma character (`,`)
- `percent` specifies the character used as a percent sign; the default value is the percent character (`%`)
- `per-mille` specifies the character used as a per mille sign; the default value is the Unicode per-mille character (#x2030)
- `zero-digit` specifies the character used as the digit zero; the default value is the digit zero (`0`)

The following attributes control the interpretation of characters in the format pattern:

- `digit` specifies the character used for a digit in the format pattern; the default value is the number sign character (`#`)
- `pattern-separator` specifies the character used to separate positive and negative sub patterns in a pattern; the default value is the semi-colon character (`;`)

The following attributes specify characters or strings that may appear in the result of formatting the number:

- `infinity` specifies the string used to represent infinity; the default value is the string `Infinity`

- `NaN` specifies the string used to represent the NaN value; the default value is the string `NaN`
- `minus-sign` specifies the character used as the default minus sign; the default value is the hyphen-minus character (-, #x2D)

12.4 Miscellaneous Additional Functions

Function: *node-set* **current**()

The **current** function returns a node-set that has the current node as its only member. For an outermost expression (an expression not occurring within another expression), the current node is always the same as the context node. Thus,

```
<xsl:value-of select="current()"/>
```

means the same as

```
<xsl:value-of select="."/>
```

However, within square brackets the current node is usually different from the context node. For example,

```
<xsl:apply-templates select="//glossary/item[@name=current()/@ref]"/>
```

will process all `item` elements that have a `glossary` parent element and that have a `name` attribute with value equal to the value of the current node's `ref` attribute. This is different from

```
<xsl:apply-templates select="//glossary/item[@name=./@ref]"/>
```

which means the same as

```
<xsl:apply-templates select="//glossary/item[@name=@ref]"/>
```

and so would process all `item` elements that have a `glossary` parent element and that have a `name` attribute and a `ref` attribute with the same value.

It is an error to use the **current** function in a pattern.

Function: *string* **unparsed-entity-uri**(*string*)

The **unparsed-entity-uri** returns the URI of the unparsed entity with the specified name in the same document as the context node (see [3.3 Unparsed Entities]). It returns the empty string if there is no such entity.

Function: *string* **generate-id**(*node-set*?)

The **generate-id** function returns a string that uniquely identifies the node in the argument node-set that is first in document order. The unique identifier must consist of ASCII alphanumeric characters and must start with an alphabetic character. Thus, the string is syntactically an XML name. An implementation is free to generate an identifier in any convenient way provided that it always generates the same identifier for the same node and that different identifiers are always generated from different nodes. An implementation is under no obligation to generate the same identifiers each time a document is transformed. There is no guarantee that a generated unique identifier will be distinct from any unique IDs specified in the source document. If the argument node-set is empty, the empty string is returned. If the argument is omitted, it defaults to the context node.

Function: *object* **system-property**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into a name using the namespace declarations in scope for the expression. The **system-property** function returns an object representing the value of the system property identified by the name. If there is no such system property, the empty string should be returned.

Implementations must provide the following system properties, which are all in the XSLT namespace:

- `xsl:version`, a number giving the version of XSLT implemented by the processor; for XSLT processors implementing the version of XSLT specified by this document, this is the number 1.0

- `xsl:vendor`, a string identifying the vendor of the XSLT processor
- `xsl:vendor-url`, a string containing a URL identifying the vendor of the XSLT processor; typically this is the host page (home page) of the vendor's Web site.

13 Messages

```
<!-- Category: instruction -->
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>
```

The `xsl:message` instruction sends a message in a way that is dependent on the XSLT processor. The content of the `xsl:message` instruction is a template. The `xsl:message` is instantiated by instantiating the content to create an XML fragment. This XML fragment is the content of the message.

> **NOTE:** An XSLT processor might implement `xsl:message` by popping up an alert box or by writing to a log file.

If the `terminate` attribute has the value `yes`, then the XSLT processor should terminate processing after sending the message. The default value is `no`.

One convenient way to do localization is to put the localized information (message text, etc.) in an XML document, which becomes an additional input file to the stylesheet. For example, suppose messages for a language `L` are stored in an XML file `resources/L.xml` in the form:

```
<messages>
  <message name="problem">A problem was detected.</message>
  <message name="error">An error was detected.</message>
</messages>
```

Then a stylesheet could use the following approach to localize messages:

```
<xsl:param name="lang" select="en"/>
<xsl:variable name="messages"
  select="document(concat('resources/', $lang, '.xml'))/messages"/>

<xsl:template name="localized-message">
  <xsl:param name="name"/>
  <xsl:message>
    <xsl:value-of select="$messages/message[@name=$name]"/>
  </xsl:message>
</xsl:template>

<xsl:template name="problem">
  <xsl:call-template name="localized-message"/>
    <xsl:with-param name="name">problem</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

14 Extensions

XSLT allows two kinds of extension, extension elements and extension functions.

This version of XSLT does not provide a mechanism for defining implementations of extensions. Therefore, an XSLT stylesheet that must be portable between XSLT implementations cannot rely on particular extensions being available. XSLT provides mechanisms that allow an XSLT stylesheet to determine whether the XSLT processor by which it is being processed has implementations of particular extensions available, and to specify what should happen if those extensions are not available. If an XSLT stylesheet is careful to make use of these mechanisms, it is possible for it to take advantage of extensions and still work with any XSLT implementation.

14.1 Extension Elements

The element extension mechanism allows namespaces to be designated as **extension namespaces**. When a namespace is designated as an extension namespace and an element with a name from that namespace occurs in a template, then the element is treated as an instruction rather than as a literal result element. The namespace determines the semantics of the instruction.

> **NOTE:** Since an element that is a child of an `xsl:stylesheet` element is not occurring *in a template*, non-XSLT top-level elements are not extension elements as defined here, and nothing in this

section applies to them.

A namespace is designated as an extension namespace by using an `extension-element-prefixes` attribute on an `xsl:stylesheet` element or an `xsl:extension-element-prefixes` attribute on a literal result element or extension element. The value of both these attributes is a whitespace-separated list of namespace prefixes. The namespace bound to each of the prefixes is designated as an extension namespace. It is an error if there is no namespace bound to the prefix on the element bearing the `extension-element-prefixes` or `xsl:extension-element-prefixes` attribute. The default namespace (as declared by `xmlns`) may be designated as an extension namespace by including `#default` in the list of namespace prefixes. The designation of a namespace as an extension namespace is effective within the subtree of the stylesheet rooted at the element bearing the `extension-element-prefixes` or `xsl:extension-element-prefixes` attribute; a subtree rooted at an `xsl:stylesheet` element does not include any stylesheets imported or included by children of that `xsl:stylesheet` element.

If the XSLT processor does not have an implementation of a particular extension element available, then the **element-available** function must return false for the name of the element. When such an extension element is instantiated, then the XSLT processor must perform fallback for the element as specified in [15 Fallback]. An XSLT processor must not signal an error merely because a template contains an extension element for which no implementation is available.

If the XSLT processor has an implementation of a particular extension element available, then the **element-available** function must return true for the name of the element.

14.2 Extension Functions

If a FunctionName in a FunctionCall expression is not an NCName (i.e. if it contains a colon), then it is treated as a call to an extension function. The FunctionName is expanded to a name using the namespace declarations from the evaluation context.

If the XSLT processor does not have an implementation of an extension function of a particular name available, then the **function-available** function must return false for that name. If such an extension function occurs in an expression and the extension function is actually called, the XSLT processor must signal an error. An XSLT processor must not signal an error merely because an expression contains an extension function for which no implementation is available.

If the XSLT processor has an implementation of an extension function of a particular name available, then the **function-available** function must return true for that name. If such an extension is called, then the XSLT processor must call the implementation passing it the function call arguments; the result returned by the implementation is returned as the result of the function call.

15 Fallback

```
<!-- Category: instruction -->
<xsl:fallback>
  <!-- Content: template -->
</xsl:fallback>
```

Normally, instantiating an `xsl:fallback` element does nothing. However, when an XSLT processor performs fallback for an instruction element, if the instruction element has one or more `xsl:fallback` children, then the content of each of the `xsl:fallback` children must be instantiated in sequence; otherwise, an error must be signaled. The content of an `xsl:fallback` element is a template.

The following functions can be used with the `xsl:choose` and `xsl:if` instructions to explicitly control how a stylesheet should behave if particular elements or functions are not available.

Function: *boolean* **element-available**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into an expanded-name using the namespace declarations in scope for the expression. The **element-available** function returns true if and only if the expanded-name is the name of an instruction. If the expanded-name has a namespace URI equal to the XSLT namespace URI, then it refers to an element defined by XSLT. Otherwise, it refers to an extension element. If the expanded-name has a null namespace URI, the **element-available** function will return false.

Function: *boolean* **function-available**(*string*)

The argument must evaluate to a string that is a QName. The QName is expanded into an expanded-name using the namespace declarations in scope for the expression. The **function-available** function returns true if and only if the expanded-name is the name of a function in the function library. If the expanded-name has a non-null namespace URI, then it refers to an extension function; otherwise, it refers to a function defined by XPath or XSLT.

16 Output

```
<!-- Category: top-level-element -->
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />
```

An XSLT processor may output the result tree as a sequence of bytes, although it is not required to be able to do so (see [17 Conformance]). The `xsl:output` element allows stylesheet authors to specify how they wish the result tree to be output. If an XSLT processor outputs the result tree, it should do so as specified by the `xsl:output` element; however, it is not required to do so.

The `xsl:output` element is only allowed as a top-level element.

The `method` attribute on `xsl:output` identifies the overall method that should be used for outputting the result tree. The value must be a QName. If the QName does not have a prefix, then it identifies a method specified in this document and must be one of `xml`, `html` or `text`. If the QName has a prefix, then the QName is expanded into an expanded-name as described in [2.4 Qualified Names]; the expanded-name identifies the output method; the behavior in this case is not specified by this document.

The default for the `method` attribute is chosen as follows. If

- the root node of the result tree has an element child,
- the expanded-name of the first element child of the root node (i.e. the document element) of the result tree has local part `html` (in any combination of upper and lower case) and a null namespace URI, and
- any text nodes preceding the first element child of the root node of the result tree contain only whitespace characters,

then the default output method is `html`; otherwise, the default output method is `xml`. The default output method should be used if there are no `xsl:output` elements or if none of the `xsl:output` elements specifies a value for the `method` attribute.

The other attributes on `xsl:output` provide parameters for the output method. The following attributes are allowed:

- `version` specifies the version of the output method
- `indent` specifies whether the XSLT processor may add additional whitespace when outputting the result tree; the value must be `yes` or `no`
- `encoding` specifies the preferred character encoding that the XSLT processor should use to encode sequences of characters as sequences of bytes; the value of the attribute should be treated case-insensitively; the value must contain only characters in the range #x21 to #x7E (i.e. printable ASCII characters); the value should either be a `charset` registered with the Internet Assigned Numbers Authority [IANA], [RFC2278] or start with `x-`
- `media-type` specifies the media type (MIME content type) of the data that results from outputting the result tree; the `charset` parameter should not be specified explicitly; instead, when the top-level media

type is `text`, a `charset` parameter should be added according to the character encoding actually used by the output method

- `doctype-system` specifies the system identifier to be used in the document type declaration
- `doctype-public` specifies the public identifier to be used in the document type declaration
- `omit-xml-declaration` specifies whether the XSLT processor should output an XML declaration; the value must be `yes` or `no`
- `standalone` specifies whether the XSLT processor should output a standalone document declaration; the value must be `yes` or `no`
- `cdata-section-elements` specifies a list of the names of elements whose text node children should be output using CDATA sections

The detailed semantics of each attribute will be described separately for each output method for which it is applicable. If the semantics of an attribute are not described for an output method, then it is not applicable to that output method.

A stylesheet may contain multiple `xsl:output` elements and may include or import stylesheets that also contain `xsl:output` elements. All the `xsl:output` elements occurring in a stylesheet are merged into a single effective `xsl:output` element. For the `cdata-section-elements` attribute, the effective value is the union of the specified values. For other attributes, the effective value is the specified value with the highest import precedence. It is an error if there is more than one such value for an attribute. An XSLT processor may signal the error; if it does not signal the error, if should recover by using the value that occurs last in the stylesheet. The values of attributes are defaulted after the `xsl:output` elements have been merged; different output methods may have different default values for an attribute.

16.1 XML Output Method

The `xml` output method outputs the result tree as a well-formed XML external general parsed entity. If the root node of the result tree has a single element node child and no text node children, then the entity should also be a well-formed XML document entity. When the entity is referenced within a trivial XML document wrapper like this

```
<!DOCTYPE doc [
<!ENTITY e SYSTEM "entity-URI">
]>
<doc>&e;</doc>
```

where *entity-URI* is a URI for the entity, then the wrapper document as a whole should be a well-formed XML document conforming to the XML Namespaces Recommendation [XML Names]. In addition, the output should be such that if a new tree was constructed by parsing the wrapper as an XML document as specified in [3 Data Model], and then removing the document element, making its children instead be children of the root node, then the new tree would be the same as the result tree, with the following possible exceptions:

- The order of attributes in the two trees may be different.
- The new tree may contain namespace nodes that were not present in the result tree.

> **NOTE:**An XSLT processor may need to add namespace declarations in the course of outputting the result tree as XML.

If the XSLT processor generated a document type declaration because of the `doctype-system` attribute, then the above requirements apply to the entity with the generated document type declaration removed.

The `version` attribute specifies the version of XML to be used for outputting the result tree. If the XSLT processor does not support this version of XML, it should use a version of XML that it does support. The version output in the XML declaration (if an XML declaration is output) should correspond to the version of XML that the processor used for outputting the result tree. The value of the `version` attribute should match the VersionNum production of the XML Recommendation [XML]. The default value is `1.0`.

The `encoding` attribute specifies the preferred encoding to use for outputting the result tree. XSLT processors

are required to respect values of `UTF-8` and `UTF-16`. For other values, if the XSLT processor does not support the specified encoding it may signal an error; if it does not signal an error it should use `UTF-8` or `UTF-16` instead. The XSLT processor must not use an encoding whose name does not match the EncName production of the XML Recommendation [XML]. If no `encoding` attribute is specified, then the XSLT processor should use either `UTF-8` or `UTF-16`. It is possible that the result tree will contain a character that cannot be represented in the encoding that the XSLT processor is using for output. In this case, if the character occurs in a context where XML recognizes character references (i.e. in the value of an attribute node or text node), then the character should be output as a character reference; otherwise (for example if the character occurs in the name of an element) the XSLT processor should signal an error.

If the `indent` attribute has the value `yes`, then the `xml` output method may output whitespace in addition to the whitespace in the result tree (possibly based on whitespace stripped from either the source document or the stylesheet) in order to indent the result nicely; if the `indent` attribute has the value `no`, it should not output any additional whitespace. The default value is `no`. The `xml` output method should use an algorithm to output additional whitespace that ensures that the result if whitespace were to be stripped from the output using the process described in [3.4 Whitespace Stripping] with the set of whitespace-preserving elements consisting of just `xsl:text` would be the same when additional whitespace is output as when additional whitespace is not output.

> **NOTE:**It is usually not safe to use `indent="yes"` with document types that include element types with mixed content.

The `cdata-section-elements` attribute contains a whitespace-separated list of QNames. Each QName is expanded into an expanded-name using the namespace declarations in effect on the `xsl:output` element in which the QName occurs; if there is a default namespace, it is used for QNames that do not have a prefix. The expansion is performed before the merging of multiple `xsl:output` elements into a single effective `xsl:output` element. If the expanded-name of the parent of a text node is a member of the list, then the text node should be output as a CDATA section. For example,

```
<xsl:output cdata-section-elements="example"/>
```

would cause a literal result element written in the stylesheet as

```
<example>&lt;foo></example>
```

or as

```
<example><![CDATA[<foo>]]></example>
```

to be output as

```
<example><![CDATA[<foo>]]></example>
```

If the text node contains the sequence of characters `]]>`, then the currently open CDATA section should be closed following the `]]` and a new CDATA section opened before the `>`. For example, a literal result element written in the stylesheet as

```
<example>]]&gt;</example>
```

would be output as

```
<example><![CDATA[]]]><![CDATA[>]]></example>
```

If the text node contains a character that is not representable in the character encoding being used to output the result tree, then the currently open CDATA section should be closed before the character, the character should be output using a character reference or entity reference, and a new CDATA section should be opened for any further characters in the text node.

CDATA sections should not be used except for text nodes that the `cdata-section-elements` attribute explicitly specifies should be output using CDATA sections.

The `xml` output method should output an XML declaration unless the `omit-xml-declaration` attribute has the value `yes`. The XML declaration should include both version information and an encoding declaration. If the `standalone` attribute is specified, it should include a standalone document declaration with the same value as the value as the value of the `standalone` attribute. Otherwise, it should not include a standalone document

declaration; this ensures that it is both a XML declaration (allowed at the beginning of a document entity) and a text declaration (allowed at the beginning of an external general parsed entity).

If the doctype-system attribute is specified, the xml output method should output a document type declaration immediately before the first element. The name following <!DOCTYPE should be the name of the first element. If doctype-public attribute is also specified, then the xml output method should output PUBLIC followed by the public identifier and then the system identifier; otherwise, it should output SYSTEM followed by the system identifier. The internal subset should be empty. The doctype-public attribute should be ignored unless the doctype-system attribute is specified.

The media-type attribute is applicable for the xml output method. The default value for the media-type attribute is text/xml.

16.2 HTML Output Method

The html output method outputs the result tree as HTML; for example,

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html"/>

<xsl:template match="/">
  <html>
   <xsl:apply-templates/>
  </html>
</xsl:template>

...

</xsl:stylesheet>
```

The version attribute indicates the version of the HTML. The default value is 4.0, which specifies that the result should be output as HTML conforming to the HTML 4.0 Recommendation [HTML].

The html output method should not output an element differently from the xml output method unless the expanded-name of the element has a null namespace URI; an element whose expanded-name has a non-null namespace URI should be output as XML. If the expanded-name of the element has a null namespace URI, but the local part of the expanded-name is not recognized as the name of an HTML element, the element should output in the same way as a non-empty, inline element such as span.

The html output method should not output an end-tag for empty elements. For HTML 4.0, the empty elements are area, base, basefont, br, col, frame, hr, img, input, isindex, link, meta and param. For example, an element written as <br/> or
</br> in the stylesheet should be output as <br>.

The html output method should recognize the names of HTML elements regardless of case. For example, elements named br, BR or Br should all be recognized as the HTML br element and output without an end-tag.

The html output method should not perform escaping for the content of the script and style elements. For example, a literal result element written in the stylesheet as

```
<script>if (a &lt; b) foo()</script>
```

or

```
<script><![CDATA[if (a < b) foo()]]></script>
```

should be output as

```
<script>if (a < b) foo()</script>
```

The html output method should not escape < characters occurring in attribute values.

If the indent attribute has the value yes, then the html output method may add or remove whitespace as it outputs the result tree, so long as it does not change how an HTML user agent would render the output. The default value is yes.

The html output method should escape non-ASCII characters in URI attribute values using the method recommended in Section B.2.1 of the HTML 4.0 Recommendation.

The html output method may output a character using a character entity reference, if one is defined for it in the version of HTML that the output method is using.

The html output method should terminate processing instructions with > rather than ?>.

The html output method should output boolean attributes (that is attributes with only a single allowed value that is equal to the name of the attribute) in minimized form. For example, a start-tag written in the stylesheet as

```
<OPTION selected="selected">
```

should be output as

```
<OPTION selected>
```

The html output method should not escape a & character occurring in an attribute value immediately followed by a { character (see Section B.7.1 of the HTML 4.0 Recommendation). For example, a start-tag written in the stylesheet as

```
<BODY bgcolor='&amp;{{randomrbg}};'>
```

should be output as

```
<BODY bgcolor='&{{randomrbg};'>
```

The encoding attribute specifies the preferred encoding to be used. If there is a HEAD element, then the html output method should add a META element immediately after the start-tag of the HEAD element specifying the character encoding actually used. For example,

```
<HEAD>
<META http-equiv="Content-Type" content="text/html; charset=EUC-JP">
...
```

It is possible that the result tree will contain a character that cannot be represented in the encoding that the XSLT processor is using for output. In this case, if the character occurs in a context where HTML recognizes character references, then the character should be output as a character entity reference or decimal numeric character reference; otherwise (for example, in a script or style element or in a comment), the XSLT processor should signal an error.

If the doctype-public or doctype-system attributes are specified, then the html output method should output a document type declaration immediately before the first element. The name following <!DOCTYPE should be HTML or html. If the doctype-public attribute is specified, then the output method should output PUBLIC followed by the specified public identifier; if the doctype-system attribute is also specified, it should also output the specified system identifier following the public identifier. If the doctype-system attribute is specified but the doctype-public attribute is not specified, then the output method should output SYSTEM followed by the specified system identifier.

The media-type attribute is applicable for the html output method. The default value is text/html.

16.3 Text Output Method

The text output method outputs the result tree by outputting the string-value of every text node in the result tree in document order without any escaping.

The media-type attribute is applicable for the text output method. The default value for the media-type attribute is text/plain.

The encoding attribute identifies the encoding that the text output method should use to convert sequences of characters to sequences of bytes. The default is system-dependent. If the result tree contains a character that cannot be represented in the encoding that the XSLT processor is using for output, the XSLT processor should signal an error.

16.4 Disabling Output Escaping

Normally, the xml output method escapes & and < (and possibly other characters) when outputting text

nodes. This ensures that the output is well-formed XML. However, it is sometimes convenient to be able to produce output that is almost, but not quite well-formed XML; for example, the output may include ill-formed sections which are intended to be transformed into well-formed XML by a subsequent non-XML aware process. For this reason, XSLT provides a mechanism for disabling output escaping. An `xsl:value-of` or `xsl:text` element may have a `disable-output-escaping` attribute; the allowed values are `yes` or `no`; the default is `no`; if the value is `yes`, then a text node generated by instantiating the `xsl:value-of` or `xsl:text` element should be output without any escaping. For example,

```
<xsl:text disable-output-escaping="yes">&lt;</xsl:text>
```

should generate the single character <.

It is an error for output escaping to be disabled for a text node that is used for something other than a text node in the result tree. Thus, it is an error to disable output escaping for an `xsl:value-of` or `xsl:text` element that is used to generate the string-value of a comment, processing instruction or attribute node; it is also an error to convert a result tree fragment to a number or a string if the result tree fragment contains a text node for which escaping was disabled. In both cases, an XSLT processor may signal the error; if it does not signal the error, it must recover by ignoring the `disable-output-escaping` attribute.

The `disable-output-escaping` attribute may be used with the `html` output method as well as with the `xml` output method. The `text` output method ignores the `disable-output-escaping` attribute, since it does not perform any output escaping.

An XSLT processor will only be able to disable output escaping if it controls how the result tree is output. This may not always be the case. For example, the result tree may be used as the source tree for another XSLT transformation instead of being output. An XSLT processor is not required to support disabling output escaping. If an `xsl:value-of` or `xsl:text` specifies that output escaping should be disabled and the XSLT processor does not support this, the XSLT processor may signal an error; if it does not signal an error, it must recover by not disabling output escaping.

If output escaping is disabled for a character that is not representable in the encoding that the XSLT processor is using for output, then the XSLT processor may signal an error; if it does not signal an error, it must recover by not disabling output escaping.

Since disabling output escaping may not work with all XSLT processors and can result in XML that is not well-formed, it should be used only when there is no alternative.

17 Conformance

A conforming XSLT processor must be able to use a stylesheet to transform a source tree into a result tree as specified in this document. A conforming XSLT processor need not be able to output the result in XML or in any other form.

> **NOTE:** Vendors of XSLT processors are strongly encouraged to provide a way to verify that their processor is behaving conformingly by allowing the result tree to be output as XML or by providing access to the result tree through a standard API such as the DOM or SAX.

A conforming XSLT processor must signal any errors except for those that this document specifically allows an XSLT processor not to signal. A conforming XSLT processor may but need not recover from any errors that it signals.

A conforming XSLT processor may impose limits on the processing resources consumed by the processing of a stylesheet.

18 Notation

The specification of each XSLT-defined element type is preceded by a summary of its syntax in the form of a model for elements of that element type. The meaning of syntax summary notation is as follows:

- An attribute is required if and only if its name is in bold.
- The string that occurs in the place of an attribute value specifies the allowed values of the attribute. If this is surrounded by curly braces, then the attribute value is treated as an attribute value template, and the string occurring within curly braces specifies the allowed values of the result of instantiating the attribute value template. Alternative allowed values are separated by `|`. A quoted string indicates a value equal to that specific string. An unquoted, italicized name specifies a particular type of value.
- If the element is allowed not to be empty, then the element contains a comment specifying the allowed content. The allowed content is specified in a similar way to an element type declaration in XML; *template* means that any mixture of text nodes, literal result elements, extension elements, and XSLT elements from the `instruction` category is allowed; *top-level-elements* means that any mixture of XSLT elements from the `top-level-element` category is allowed.
- The element is prefaced by comments indicating if it belongs to the `instruction` category or `top-level-element` category or both. The category of an element just affects whether it is allowed in the content of elements that allow a *template* or *top-level-elements*.

A References

A.1 Normative References

XML
World Wide Web Consortium. *Extensible Markup Language (XML) 1.0*. W3C Recommendation. See http://www.w3.org/TR/1998/REC-xml-19980210

XML Names
World Wide Web Consortium. *Namespaces in XML*. W3C Recommendation. See http://www.w3.org/TR/REC-xml-names

XPath
World Wide Web Consortium. *XML Path Language*. W3C Recommendation. See http://www.w3.org/TR/xpath

A.2 Other References

CSS2
World Wide Web Consortium. *Cascading Style Sheets, level 2 (CSS2)*. W3C Recommendation. See http://www.w3.org/TR/1998/REC-CSS2-19980512

DSSSL
International Organization for Standardization, International Electrotechnical Commission. *ISO/IEC 10179:1996. Document Style Semantics and Specification Language (DSSSL)*. International Standard.

HTML
World Wide Web Consortium. *HTML 4.0 specification*. W3C Recommendation. See http://www.w3.org/TR/REC-html40

IANA
Internet Assigned Numbers Authority. *Character Sets*. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

RFC2278
N. Freed, J. Postel. *IANA Charset Registration Procedures*. IETF RFC 2278. See http://www.ietf.org/rfc/rfc2278.txt.

RFC2376
E. Whitehead, M. Murata. *XML Media Types*. IETF RFC 2376. See http://www.ietf.org/rfc/rfc2376.txt.

RFC2396
T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF RFC 2396. See http://www.ietf.org/rfc/rfc2396.txt.

UNICODE TR10
Unicode Consortium. *Unicode Technical Report #10. Unicode Collation Algorithm*. Unicode Technical Report. See http://www.unicode.org/unicode/reports/tr10/index.html.

XHTML
World Wide Web Consortium. *XHTML 1.0: The Extensible HyperText Markup Language*. W3C Proposed Recommendation. See http://www.w3.org/TR/xhtml1

XPointer
World Wide Web Consortium. *XML Pointer Language (XPointer)*. W3C Working Draft. See http://www.w3.org/TR/xptr

XML Stylesheet

World Wide Web Consortium. *Associating stylesheets with XML documents*. W3C Recommendation. See http://www.w3.org/TR/xml-stylesheet

XSL

World Wide Web Consortium. *Extensible Stylesheet Language (XSL)*. W3C Working Draft. See http://www.w3.org/TR/WD-xsl

B Element Syntax Summary

```
<!-- Category: instruction -->
<xsl:apply-imports />

<!-- Category: instruction -->
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>

<!-- Category: instruction -->
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }>
  <!-- Content: template -->
</xsl:attribute>

<!-- Category: top-level-element -->
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>

<!-- Category: instruction -->
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>

<!-- Category: instruction -->
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>

<!-- Category: instruction -->
<xsl:comment>
  <!-- Content: template -->
</xsl:comment>

<!-- Category: instruction -->
<xsl:copy
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>

<!-- Category: instruction -->
<xsl:copy-of
  select = expression />

<!-- Category: top-level-element -->
<xsl:decimal-format
  name = qname
  decimal-separator = char
  grouping-separator = char
  infinity = string
  minus-sign = char
  NaN = string
  percent = char
  per-mille = char
  zero-digit = char
  digit = char
  pattern-separator = char />

<!-- Category: instruction -->
<xsl:element
  name = { qname }
  namespace = { uri-reference }
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>

<!-- Category: instruction -->
<xsl:fallback>
  <!-- Content: template -->
</xsl:fallback>

<!-- Category: instruction -->
<xsl:for-each
```



```
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>

<!-- Category: instruction -->
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>

<xsl:import
  href = uri-reference />

<!-- Category: top-level-element -->
<xsl:include
  href = uri-reference />

<!-- Category: top-level-element -->
<xsl:key
  name = qname
  match = pattern
  use = expression />

<!-- Category: instruction -->
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>

<!-- Category: top-level-element -->
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />

<!-- Category: instruction -->
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />

<xsl:otherwise>
  <!-- Content: template -->
</xsl:otherwise>

<!-- Category: top-level-element -->
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />

<!-- Category: top-level-element -->
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>

<!-- Category: top-level-element -->
<xsl:preserve-space
  elements = tokens />

<!-- Category: instruction -->
<xsl:processing-instruction
  name = { ncname }>
  <!-- Content: template -->
</xsl:processing-instruction>

<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" | qname-but-not-ncname }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />

<!-- Category: top-level-element -->
<xsl:strip-space
  elements = tokens />
```

```
<xsl:stylesheet
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:stylesheet>

<!-- Category: top-level-element -->
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- Content: (xsl:param*, template) -->
</xsl:template>

<!-- Category: instruction -->
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>

<xsl:transform
  id = id
  extension-element-prefixes = tokens
  exclude-result-prefixes = tokens
  version = number>
  <!-- Content: (xsl:import*, top-level-elements) -->
</xsl:transform>

<!-- Category: instruction -->
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />

<!-- Category: top-level-element -->
<!-- Category: instruction -->
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>

<xsl:when
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>

<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

C DTD Fragment for XSLT Stylesheets (Non-Normative)

> **NOTE:** This DTD Fragment is not normative because XML 1.0 DTDs do not support XML Namespaces and thus cannot correctly describe the allowed structure of an XSLT stylesheet.

The following entity can be used to construct a DTD for XSLT stylesheets that create instances of a particular result DTD. Before referencing the entity, the stylesheet DTD must define a `result-elements` parameter entity listing the allowed result element types. For example:

```
<!ENTITY % result-elements "
  | fo:inline-sequence
  | fo:block
">
```

Such result elements should be declared to have `xsl:use-attribute-sets` and `xsl:extension-element-prefixes` attributes. The following entity declares the `result-element-atts` parameter for this purpose. The content that XSLT allows for result elements is the same as it allows for the XSLT elements that are declared in the following entity with a content model of `%template;`. The DTD may use a more restrictive content model than `%template;` to reflect the constraints of the result DTD.

The DTD may define the `non-xsl-top-level` parameter entity to allow additional top-level elements from namespaces other than the XSLT namespace.

The use of the `xsl:` prefix in this DTD does not imply that XSLT stylesheets are required to use this prefix. Any of the elements declared in this DTD may have attributes whose name starts with `xmlns:` or is equal to

`xmlns` in addition to the attributes declared in this DTD.

```
<!ENTITY % char-instructions "
  | xsl:apply-templates
  | xsl:call-template
  | xsl:apply-imports
  | xsl:for-each
  | xsl:value-of
  | xsl:copy-of
  | xsl:number
  | xsl:choose
  | xsl:if
  | xsl:text
  | xsl:copy
  | xsl:variable
  | xsl:message
  | xsl:fallback
">

<!ENTITY % instructions "
  %char-instructions;
  | xsl:processing-instruction
  | xsl:comment
  | xsl:element
  | xsl:attribute
">

<!ENTITY % char-template "
 (#PCDATA
  %char-instructions;)*
">

<!ENTITY % template "
 (#PCDATA
  %instructions;
  %result-elements;)*
">

<!-- Used for the type of an attribute value that is a URI reference.-->
<!ENTITY % URI "CDATA">

<!-- Used for the type of an attribute value that is a pattern.-->
<!ENTITY % pattern "CDATA">

<!-- Used for the type of an attribute value that is an
     attribute value template.-->
<!ENTITY % avt "CDATA">

<!-- Used for the type of an attribute value that is a QName; the prefix
     gets expanded by the XSLT processor. -->
<!ENTITY % qname "NMTOKEN">

<!-- Like qname but a whitespace-separated list of QNames. -->
<!ENTITY % qnames "NMTOKENS">

<!-- Used for the type of an attribute value that is an expression.-->
<!ENTITY % expr "CDATA">

<!-- Used for the type of an attribute value that consists
     of a single character.-->
<!ENTITY % char "CDATA">

<!-- Used for the type of an attribute value that is a priority. -->
<!ENTITY % priority "NMTOKEN">

<!ENTITY % space-att "xml:space (default|preserve) #IMPLIED">

<!-- This may be overridden to customize the set of elements allowed
at the top-level. -->

<!ENTITY % non-xsl-top-level "">

<!ENTITY % top-level "
 (xsl:import*,
  (xsl:include
  | xsl:strip-space
  | xsl:preserve-space
  | xsl:output
  | xsl:key
  | xsl:decimal-format
  | xsl:attribute-set
  | xsl:variable
  | xsl:param
  | xsl:template
  | xsl:namespace-alias
  %non-xsl-top-level;)*)
">

<!ENTITY % top-level-atts '
  extension-element-prefixes CDATA #IMPLIED
  exclude-result-prefixes CDATA #IMPLIED
  id ID #IMPLIED
  version NMTOKEN #REQUIRED
  xmlns:xsl CDATA #FIXED "http://www.w3.org/1999/XSL/Transform"
  %space-att;
```

```
'>

<!-- This entity is defined for use in the ATTLIST declaration
for result elements. -->

<!ENTITY % result-element-atts '
  xsl:extension-element-prefixes CDATA #IMPLIED
  xsl:exclude-result-prefixes CDATA #IMPLIED
  xsl:use-attribute-sets %qnames; #IMPLIED
  xsl:version NMTOKEN #IMPLIED
'>

<!ELEMENT xsl:stylesheet %top-level;>
<!ATTLIST xsl:stylesheet %top-level-atts;>

<!ELEMENT xsl:transform %top-level;>
<!ATTLIST xsl:transform %top-level-atts;>

<!ELEMENT xsl:import EMPTY>
<!ATTLIST xsl:import href %URI; #REQUIRED>

<!ELEMENT xsl:include EMPTY>
<!ATTLIST xsl:include href %URI; #REQUIRED>

<!ELEMENT xsl:strip-space EMPTY>
<!ATTLIST xsl:strip-space elements CDATA #REQUIRED>

<!ELEMENT xsl:preserve-space EMPTY>
<!ATTLIST xsl:preserve-space elements CDATA #REQUIRED>

<!ELEMENT xsl:output EMPTY>
<!ATTLIST xsl:output
  method %qname; #IMPLIED
  version NMTOKEN #IMPLIED
  encoding CDATA #IMPLIED
  omit-xml-declaration (yes|no) #IMPLIED
  standalone (yes|no) #IMPLIED
  doctype-public CDATA #IMPLIED
  doctype-system CDATA #IMPLIED
  cdata-section-elements %qnames; #IMPLIED
  indent (yes|no) #IMPLIED
  media-type CDATA #IMPLIED
>

<!ELEMENT xsl:key EMPTY>
<!ATTLIST xsl:key
  name %qname; #REQUIRED
  match %pattern; #REQUIRED
  use %expr; #REQUIRED
>

<!ELEMENT xsl:decimal-format EMPTY>
<!ATTLIST xsl:decimal-format
  name %qname; #IMPLIED
  decimal-separator %char; "."
  grouping-separator %char; ","
  infinity CDATA "Infinity"
  minus-sign %char; "-"
  NaN CDATA "NaN"
  percent %char; "%"
  per-mille %char; "&#x2030;"
  zero-digit %char; "0"
  digit %char; "#"
  pattern-separator %char; ";"
>

<!ELEMENT xsl:namespace-alias EMPTY>
<!ATTLIST xsl:namespace-alias
  stylesheet-prefix CDATA #REQUIRED
  result-prefix CDATA #REQUIRED
>

<!ELEMENT xsl:template
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:param)*
>

<!ATTLIST xsl:template
  match %pattern; #IMPLIED
  name %qname; #IMPLIED
  priority %priority; #IMPLIED
  mode %qname; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:value-of EMPTY>
<!ATTLIST xsl:value-of
  select %expr; #REQUIRED
  disable-output-escaping (yes|no) "no"
>

<!ELEMENT xsl:copy-of EMPTY>
<!ATTLIST xsl:copy-of select %expr; #REQUIRED>
```



```
<!ELEMENT xsl:number EMPTY>
<!ATTLIST xsl:number
   level (single|multiple|any) "single"
   count %pattern; #IMPLIED
   from %pattern; #IMPLIED
   value %expr; #IMPLIED
   format %avt; '1'
   lang %avt; #IMPLIED
   letter-value %avt; #IMPLIED
   grouping-separator %avt; #IMPLIED
   grouping-size %avt; #IMPLIED
>

<!ELEMENT xsl:apply-templates (xsl:sort|xsl:with-param)*>
<!ATTLIST xsl:apply-templates
  select %expr; "node()"
  mode %qname; #IMPLIED
>

<!ELEMENT xsl:apply-imports EMPTY>

<!-- xsl:sort cannot occur after any other elements or
any non-whitespace character -->

<!ELEMENT xsl:for-each
 (#PCDATA
  %instructions;
  %result-elements;
  | xsl:sort)*
>

<!ATTLIST xsl:for-each
  select %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:sort EMPTY>
<!ATTLIST xsl:sort
  select %expr; "."
  lang %avt; #IMPLIED
  data-type %avt; "text"
  order %avt; "ascending"
  case-order %avt; #IMPLIED
>

<!ELEMENT xsl:if %template;>
<!ATTLIST xsl:if
  test %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:choose (xsl:when+, xsl:otherwise?)>
<!ATTLIST xsl:choose %space-att;>

<!ELEMENT xsl:when %template;>
<!ATTLIST xsl:when
  test %expr; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:otherwise %template;>
<!ATTLIST xsl:otherwise %space-att;>

<!ELEMENT xsl:attribute-set (xsl:attribute)*>
<!ATTLIST xsl:attribute-set
  name %qname; #REQUIRED
  use-attribute-sets %qnames; #IMPLIED
>

<!ELEMENT xsl:call-template (xsl:with-param)*>
<!ATTLIST xsl:call-template
  name %qname; #REQUIRED
>

<!ELEMENT xsl:with-param %template;>
<!ATTLIST xsl:with-param
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:variable %template;>
<!ATTLIST xsl:variable
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:param %template;>
<!ATTLIST xsl:param
  name %qname; #REQUIRED
  select %expr; #IMPLIED
>

<!ELEMENT xsl:text (#PCDATA)>
<!ATTLIST xsl:text
  disable-output-escaping (yes|no) "no"
>
```

```
<!ELEMENT xsl:processing-instruction %char-template;>
<!ATTLIST xsl:processing-instruction
  name %avt; #REQUIRED
  %space-att;
>

<!ELEMENT xsl:element %template;>
<!ATTLIST xsl:element
  name %avt; #REQUIRED
  namespace %avt; #IMPLIED
  use-attribute-sets %qnames; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:attribute %char-template;>
<!ATTLIST xsl:attribute
  name %avt; #REQUIRED
  namespace %avt; #IMPLIED
  %space-att;
>

<!ELEMENT xsl:comment %char-template;>
<!ATTLIST xsl:comment %space-att;>

<!ELEMENT xsl:copy %template;>
<!ATTLIST xsl:copy
  %space-att;
  use-attribute-sets %qnames; #IMPLIED
>

<!ELEMENT xsl:message %template;>
<!ATTLIST xsl:message
  %space-att;
  terminate (yes|no) "no"
>

<!ELEMENT xsl:fallback %template;>
<!ATTLIST xsl:fallback %space-att;>
```

D Examples (Non-Normative)

D.1 Document Example

This example is a stylesheet for transforming documents that conform to a simple DTD into XHTML [XHTML].
The DTD is:

```
<!ELEMENT doc (title, chapter*)>
<!ELEMENT chapter (title, (para|note)*, section*)>
<!ELEMENT section (title, (para|note)*)>
<!ELEMENT title (#PCDATA|emph)*>
<!ELEMENT para (#PCDATA|emph)*>
<!ELEMENT note (#PCDATA|emph)*>
<!ELEMENT emph (#PCDATA|emph)*>
```

The stylesheet is:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/TR/xhtml1/strict">

<xsl:strip-space elements="doc chapter section"/>
<xsl:output
   method="xml"
   indent="yes"
   encoding="iso-8859-1"
/>

<xsl:template match="doc">
 <html>
   <head>
     <title>
       <xsl:value-of select="title"/>
     </title>
   </head>
   <body>
     <xsl:apply-templates/>
   </body>
 </html>
</xsl:template>

<xsl:template match="doc/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>

<xsl:template match="chapter/title">
  <h2>
    <xsl:apply-templates/>
  </h2>
</xsl:template>

<xsl:template match="section/title">
  <h3>
    <xsl:apply-templates/>
  </h3>
</xsl:template>

<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>

<xsl:template match="note">
  <p class="note">
    <b>NOTE: </b>
    <xsl:apply-templates/>
  </p>
</xsl:template>

<xsl:template match="emph">
  <em>
    <xsl:apply-templates/>
  </em>
</xsl:template>

</xsl:stylesheet>
```

With the following input document

```
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
<title>Document Title</title>
<chapter>
<title>Chapter Title</title>
<section>
<title>Section Title</title>
<para>This is a test.</para>
<note>This is a note.</note>
</section>
<section>
<title>Another Section Title</title>
<para>This is <emph>another</emph> test.</para>
<note>This is another note.</note>
</section>
</chapter>
</doc>
```

it would produce the following result

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head>
<title>Document Title</title>
</head>
<body>
<h1>Document Title</h1>
<h2>Chapter Title</h2>
<h3>Section Title</h3>
<p>This is a test.</p>
<p class="note">
<b>NOTE: </b>This is a note.</p>
<h3>Another Section Title</h3>
<p>This is <em>another</em> test.</p>
<p class="note">
<b>NOTE: </b>This is another note.</p>
</body>
</html>
```

D.2 Data Example

This is an example of transforming some data represented in XML using three different XSLT stylesheets to produce three different representations of the data, HTML, SVG and VRML.

The input data is:

```
<sales>

        <division id="North">
                <revenue>10</revenue>
                <growth>9</growth>
                <bonus>7</bonus>
        </division>

        <division id="South">
                <revenue>4</revenue>
                <growth>3</growth>
```

```
                <bonus>4</bonus>
        </division>

        <division id="West">
                <revenue>6</revenue>
                <growth>-1.5</growth>
                <bonus>2</bonus>
        </division>

</sales>
```

The following stylesheet, which uses the simplified syntax described in [2.3 Literal Result Element as Stylesheet], transforms the data into HTML:

```
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      lang="en">
    <head>
        <title>Sales Results By Division</title>
    </head>
    <body>
        <table border="1">
            <tr>
                <th>Division</th>
                <th>Revenue</th>
                <th>Growth</th>
                <th>Bonus</th>
            </tr>
            <xsl:for-each select="sales/division">
                <!-- order the result by revenue -->
                <xsl:sort select="revenue"
                          data-type="number"
                          order="descending"/>
                <tr>
                    <td>
                        <em><xsl:value-of select="@id"/></em>
                    </td>
                    <td>
                        <xsl:value-of select="revenue"/>
                    </td>
                    <td>
                        <!-- highlight negative growth in red -->
                        <xsl:if test="growth &lt; 0">
                            <xsl:attribute name="style">
                                <xsl:text>color:red</xsl:text>
                            </xsl:attribute>
                        </xsl:if>
                        <xsl:value-of select="growth"/>
                    </td>
                    <td>
                        <xsl:value-of select="bonus"/>
                    </td>
                </tr>
            </xsl:for-each>
        </table>
    </body>
</html>
```

The HTML output is:

```
<html lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<title>Sales Results By Division</title>
</head>
<body>
<table border="1">
<tr>
<th>Division</th><th>Revenue</th><th>Growth</th><th>Bonus</th>
</tr>
<tr>
<td><em>North</em></td><td>10</td><td>9</td><td>7</td>
</tr>
<tr>
<td><em>West</em></td><td>6</td><td style="color:red">-1.5</td><td>2</td>
</tr>
<tr>
<td><em>South</em></td><td>4</td><td>3</td><td>4</td>
</tr>
</table>
</body>
</html>
```

The following stylesheet transforms the data into SVG:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/Graphics/SVG/SVG-19990812.dtd">

<xsl:output method="xml" indent="yes" media-type="image/svg"/>
```



```
<xsl:template match="/">

<svg width = "3in" height="3in">
    <g style = "stroke: #000000">
        <!-- draw the axes -->
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
        <xsl:for-each select="sales/division">
            <!-- define some useful variables -->

            <!-- the bar's x position -->
            <xsl:variable name="pos"
                          select="(position()*40)-30"/>

            <!-- the bar's height -->
            <xsl:variable name="height"
                          select="revenue*10"/>

            <!-- the rectangle -->
            <rect x="{$pos}" y="{150-$height}"
                  width="20" height="{$height}"/>

            <!-- the text label -->
            <text x="{$pos}" y="165">
                <xsl:value-of select="@id"/>
            </text>

            <!-- the bar value -->
            <text x="{$pos}" y="{145-$height}">
                <xsl:value-of select="revenue"/>
            </text>
        </xsl:for-each>
    </g>
</svg>

</xsl:template>
</xsl:stylesheet>
```

The SVG output is:

```
<svg width="3in" height="3in"
     xmlns="http://www.w3.org/Graphics/SVG/svg-19990412.dtd">
    <g style="stroke: #000000">
        <line x1="0" x2="150" y1="150" y2="150"/>
        <line x1="0" x2="0" y1="0" y2="150"/>
        <text x="0" y="10">Revenue</text>
        <text x="150" y="165">Division</text>
        <rect x="10" y="50" width="20" height="100"/>
        <text x="10" y="165">North</text>
        <text x="10" y="45">10</text>
        <rect x="50" y="110" width="20" height="40"/>
        <text x="50" y="165">South</text>
        <text x="50" y="105">4</text>
        <rect x="90" y="90" width="20" height="60"/>
        <text x="90" y="165">West</text>
        <text x="90" y="85">6</text>
    </g>
</svg>
```

The following stylesheet transforms the data into VRML:

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<!-- generate text output as mime type model/vrml, using default charset -->
<xsl:output method="text" encoding="UTF-8" media-type="model/vrml"/>

        <xsl:template match="/">#VRML V2.0 utf8

# externproto definition of a single bar element
EXTERNPROTO bar [
  field SFInt32 x
  field SFInt32 y
  field SFInt32 z
  field SFString name
  ]
  "http://www.vrml.org/WorkingGroups/dbwork/barProto.wrl"

# inline containing the graph axes
Inline {
        url "http://www.vrml.org/WorkingGroups/dbwork/barAxes.wrl"
        }

                <xsl:for-each select="sales/division">
bar {
        x <xsl:value-of select="revenue"/>
        y <xsl:value-of select="growth"/>
        z <xsl:value-of select="bonus"/>
        name "<xsl:value-of select="@id"/>"
        }
                </xsl:for-each>
```

```
        </xsl:template>

</xsl:stylesheet>
```

The VRML output is:

```
#VRML V2.0 utf8

# externproto definition of a single bar element
EXTERNPROTO bar [
  field SFInt32 x
  field SFInt32 y
  field SFInt32 z
  field SFString name
  ]
  "http://www.vrml.org/WorkingGroups/dbwork/barProto.wrl"

# inline containing the graph axes
Inline {
        url "http://www.vrml.org/WorkingGroups/dbwork/barAxes.wrl"
        }


bar {
        x 10
        y 9
        z 7
        name "North"
        }

bar {
        x 4
        y 3
        z 4
        name "South"
        }

bar {
        x 6
        y -1.5
        z 2
        name "West"
        }
```

E Acknowledgements (Non-Normative)

The following have contributed to authoring this draft:

- Daniel Lipkin, Saba
- Jonathan Marsh, Microsoft
- Henry Thompson, University of Edinburgh
- Norman Walsh, Arbortext
- Steve Zilles, Adobe

This specification was developed and approved for publication by the W3C XSL Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current members of the XSL WG are:

Sharon Adler, IBM (Co-Chair); Anders Berglund, IBM; Perin Blanchard, Novell; Scott Boag, Lotus; Larry Cable, Sun; Jeff Caruso, Bitstream; James Clark; Peter Danielsen, Bell Labs; Don Day, IBM; Stephen Deach, Adobe; Dwayne Dicks, SoftQuad; Andrew Greene, Bitstream; Paul Grosso, Arbortext; Eduardo Gutentag, Sun; Juliane Harbarth, Software AG; Mickey Kimchi, Enigma; Chris Lilley, W3C; Chris Maden, Exemplary Technologies; Jonathan Marsh, Microsoft; Alex Milowski, Lexica; Steve Muench, Oracle; Scott Parnell, Xerox; Vincent Quint, W3C; Dan Rapp, Novell; Gregg Reynolds, Datalogics; Jonathan Robie, Software AG; Mark Scardina, Oracle; Henry Thompson, University of Edinburgh; Philip Wadler, Bell Labs; Norman Walsh, Arbortext; Sanjiva Weerawarana, IBM; Steve Zilles, Adobe (Co-Chair)

F Changes from Proposed Recommendation (Non-Normative)

The following are the changes since the Proposed Recommendation:

- The `xsl:version` attribute is required on a literal result element used as a stylesheet (see [2.3 Literal Result Element as Stylesheet]).
- The `data-type` attribute on `xsl:sort` can use a prefixed name to specify a data-type not defined by XSLT (see [10 Sorting]).

G Features under Consideration for Future Versions of XSLT (Non-Normative)

The following features are under consideration for versions of XSLT after XSLT 1.0:

- a conditional expression;
- support for XML Schema datatypes and archetypes;
- support for something like style rules in the original XSL submission;
- an attribute to control the default namespace for names occurring in XSLT attributes;
- support for entity references;
- support for DTDs in the data model;
- support for notations in the data model;
- a way to get back from an element to the elements that reference it (e.g. by IDREF attributes);
- an easier way to get an ID or key in another document;
- support for regular expressions for matching against any or all of text nodes, attribute values, attribute names, element type names;
- case-insensitive comparisons;
- normalization of strings before comparison, for example for compatibility characters;
- a function `string resolve(node-set)` function that treats the value of the argument as a relative URI and turns it into an absolute URI using the base URI of the node;
- multiple result documents;
- defaulting the `select` attribute on `xsl:value-of` to the current node;
- an attribute on `xsl:attribute` to control how the attribute value is normalized;
- additional attributes on `xsl:sort` to provide further control over sorting, such as relative order of scripts;
- a way to put the text of a resource identified by a URI into the result tree;
- allow unions in steps (e.g. `foo/(bar|baz)`);
- allow for result tree fragments all operations that are allowed for node-sets;
- a way to group together consecutive nodes having duplicate subelements or attributes;
- features to make handling of the HTML `style` attribute more convenient.